Программирование на языке Pascal

(основы)

Мещанинов Н. А.

2010

Программирование на языке Pascal

Модуль 1. Введение в язык Pascal

Что такое программирование?

Программирование – это процесс написания программ при помощи *языков программирования*. Синтаксис (т.е. используемые служебные слова, команды и правила их применения) современных языков программирования, как правило, состоит из словесных конструкций, приближенных к естественной английской речи[1]. Например, если требуется найти максимальное из чисел `a` и `b`, то фрагмент когда может быть примерно следующим:

Листинг 1

```
if a>b then
  c:=a
else
  c:=b;
```

На русском языке такая конструкция читается так: «**Если** число `a` больше числа `b` **тогда** число `c` сделать равным числу `a`, **в противном случае** число `c` сделать равным числу `b`». Не пугайтесь, если какие-то символы для вас пока непонятны. В дальнейшем, каждый из них будет подробно описан.

Совершенно очевидно, что центральный процессор компьютера «не умеет» понимать конструкции, наподобие той, что мы привели выше. Единственные команды, которые выполняет процессор – это машинные команды (машинные коды, native codes), которые специфичны для каждого из процессоров. На заре компьютерной эры, программисты писали только при помощи этих команд или (уже попозже) при помощи языка *ассемблера*, конструкции которого однозначно соответствуют командам процессора. Написание программ в этом случае становится очень трудоемким и сложным процессом. Код программы сложно воспринимается и требуется намного больше времени для поиска ошибок. Взгляните на код, приведенный ниже.

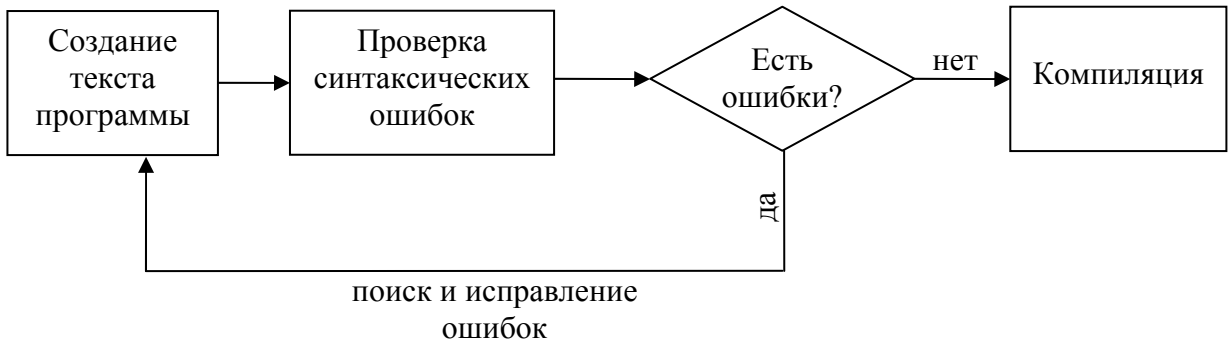
Листинг 2

```
mov ax,a
mov bx,b
cmp ax,bx
jc @els
mov c,ax
jmp @end
@els:
mov c,bx
@end:
```

[1] Есть, конечно, шуточные исключения, созданные только ради того, чтобы максимально усложнить восприятие текста программы, но подобного рода языки не используются для написания серьезных программ. Примером такого эзотерического языка является язык Whitespace, конструкции которого целиком состоят из символов, никак не отображаемых в текстовых редакторах – символы табуляции, пробела и перевода каретки.

Этот код выполняет те же функции, что и код в Листинг 1 однако, согласитесь, его сложнее воспринимать, и выглядит он менее понятно.
Разумеется, раз программисты пишут на непонятном для процессора языке, необходимо каким-то образом «переводить» для него текст программы. Этим занимается специальный модуль, который называется *компилятор*. В его обязанности как раз и входит перевод текста программы с удобного для программиста языка, как например Pascal, в понятные для процессора команды. Этот процесс, называемый компиляцией, в случае отсутствия каких-либо ошибок, завершается созданием исполняемого файла (в операционных системах DOS и Windows – это EXE файлы или файлы динамически подключаемых библиотек DLL). Компиляция практически полностью спрятана от наших глаз, и мы не будем в него вмешиваться (за редким исключением). Наша задача – научиться грамотно и правильно писать программы на языке Pascal.
Сам процесс создания программ, с нашей точки зрения будет выглядеть, как показано на рисунке ниже.

Рисунок 1



Среда программирования Turbo Pascal 7

Несмотря на то, что текст программы можно создавать в любом текстовом редакторе, как например «Блокнот», обычно программы создаются в специализированных средах (IDE[2]), которые помимо текстового редактора содержат набор утилит, помогающих находить и исправлять ошибки, встроенный компилятор и другие полезные для программистов модули.
Среда программирования, в которой мы будем работать, несмотря на то, что довольно старая (последняя версия датирована 1992 годом), заложила концепции построения IDE, которым до сих пор руководствуются создатели современных сред разработки. Внешний вид среды изображен на рисунке 2.
На этом рисунке цифрами обозначены:

1. главное меню;
2. редактор исходного текста программы;
3. окно просмотра значений используемых программой переменных.

Главное меню содержит следующие пункты:

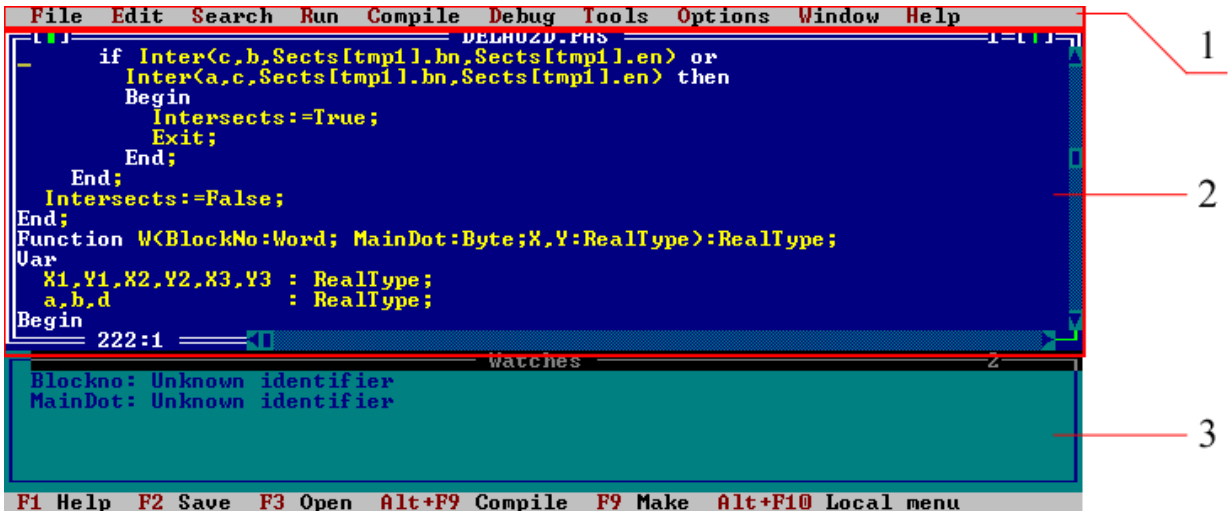
- File – содержит основные операции по управлению файлами исходного кода, такие как создание, открытие и сохранение файлов. Файлы исходного кода сохраняются в обычном текстовом файле с расширением PAS.
- Edit – содержит основные операции по редактированию исходного кода, таких как копирование, вырезание и вставку фрагментов исходного текста, а также операции по отмене действий (Undo, Redo). Сочетания клавиш отличаются от привычных

[2] Integrated Development Environment – интегрированная среда разработки.

Ctrl+C, Ctrl+V и Ctrl+X, Ctrl+Z, к которым привыкли пользователи операционной системы Windows. Вместо них используется Shift+Ins, Ctrl+Ins для вставки и копирования соответственно, Shift+Del для вырезания, Alt+Backspace – для отмены действий.

– Search – содержит операции по поиску и замене фрагментов текста.

Рисунок 2



– Run – позволяет запустить программу, предварительно откомпилировав ее, а также содержит средства для пошагового выполнения, что удобно в некоторых случаях при поисках ошибок в исходном коде. При пошаговом выполнении (Trace into) Turbo Pascal подсвечивает строчку кода, которая будет выполнена при очередном нажатии на соответствующий пункт меню (хотя чаще используют кнопку F7). Со средствами пошагового выполнения и отладки мы познакомимся чуть позже. Тогда же и рассмотрим подробнее элементы другого меню – Debug (отладка).

– Compile – позволяет компилировать программу и содержит пункты, управляющие режимами компиляции.

– Tools – содержит пункты, вызывающие дополнительные инструменты, такие как Turbo Assembler, Turbo Debugger и др.

– Options – объединяет в себе пункты вызова диалоговых окон настроек компилятора и внешнего вида среды программирования.

– Window – позволяет управлять расположением окон на рабочем пространстве и переключаться между ними.

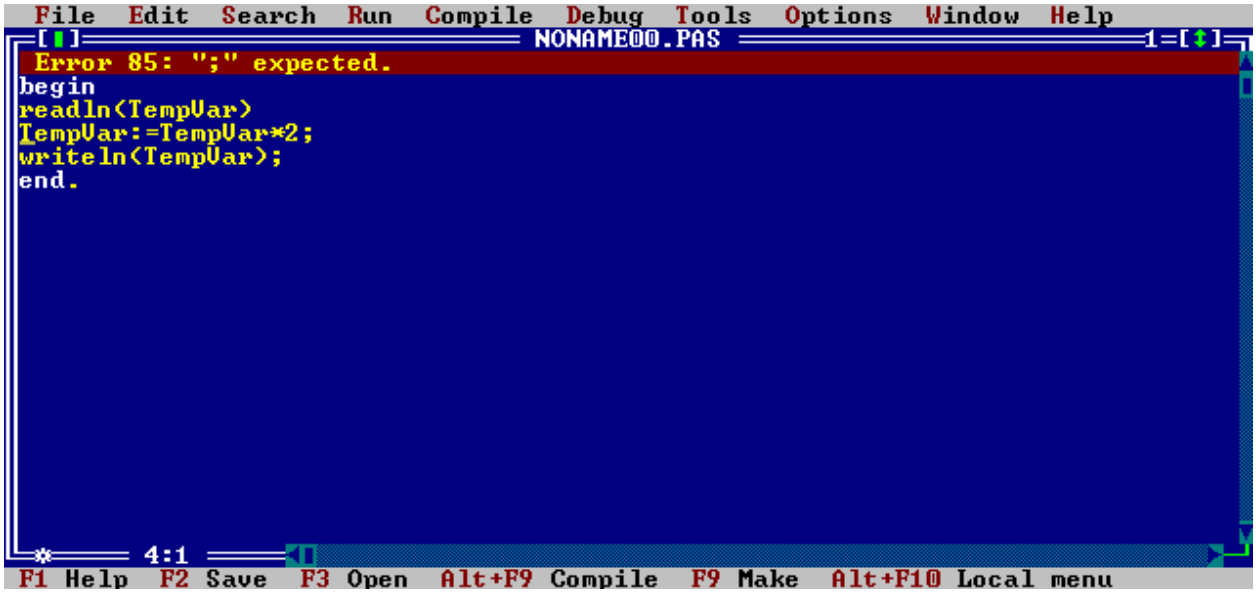
– Help – позволяет отобразить окно справочной системы или выбрать раздел справки.

Редактор исходного кода обладает функцией подсветки синтаксиса. Цвета подсветки могут быть определены свои, при помощи меню Options->Environment->Colors->Syntax. По умолчанию, все *ключевые слова* (т.е. слова, обладающие специальным значением, например определяющими начало блока кода или объявляющие какой-либо раздел в программе) выделяются белым цветом (см. рисунок 2).

При попытке компиляции программы (обычно для этого используется клавиша F9) или при ее запуске (Ctrl+F9), в случае обнаружения *синтаксическим анализатором*[3] ошибок, курсор будет мигать на месте в строке, содержащем ошибку, а в верхней либо в нижней части экрана будет выводиться сообщение об ошибке, как показано на рисунке 3.

[3] Модуль, который проверят правильность написания текста программы, основываясь на правилах языка

Рисунок 3



Переменные и типы данных

Что такое переменная?

Любые данные, с которыми должна работать программа, должны располагаться в оперативной памяти. Для того, чтобы получить доступ к этим данным, необходимо обратиться к соответствующей ячейке оперативной памяти, каждая из которых имеет свой номер – т.е. *адрес.* Адрес не является постоянным, а каждый раз изменяется при запуске программы, в зависимости от того, какие участки оперативной памяти в данный момент свободны и от некоторых других причин. Чтобы освободить программиста от необходимости обращаться к данным по адресу в памяти, было введено понятие *переменной*. Переменная – это именованный участок в памяти. Т.е программисту достаточно придумать имя для какого-нибудь значения и потом работать с этим значением, используя это имя. Перед тем как работать с переменными, необходимо их *объявить,* чтобы под них было выделено место в оперативной памяти. Все переменные объявляются в специальном месте программного кода, называемом *разделом деклараций.* Рассмотрим описанные выше аспекты с точки зрения программиста и с точки зрения процессора на примере программы, с которой мы уже сталкивались в начале этого модуля – определения наибольшего числа.

С точки зрения программиста	**С точки зрения процессора**
Объявляю три переменные `a`, `b` и `c`.	**Выделяю память для трех чисел:** – **Для переменной `a` – это будет ячейка с адресом `0x0b892a98`**[4] – **Для переменной `b` – это будет ячейка с адресом `0xb892a9a`**

[4] Не пугайтесь, для компактности и для удобства адреса памяти записывают в шестнадцатеричной системе исчисления – это такая система, в которой используются не десять цифр, а шестнадцать. Помимо цифр 0..9 вводятся буквы-цифры A, B, C, D, E, F. В приведенном примере адрес памяти 0b892a98 можно было записать и в более привычном, десятичном виде – 193538712. Приставка «0x» перед числом просто предупреждает о том, что дальше идет запись в шестнадцатеричной системе.

– **Для переменной `c` – это будет ячейка с адресом `0xb892a9c`**

Записываю в переменную `a` значение, равное 10.
Записываю в переменную `b` значение, равное 15.

Записываю по адресу `0x0b892a98` значение 10.
Записываю по адресу `0xb892a9a` значение 15.

Если число `a` больше `b`, тогда в переменную с записать значение переменной `a`, в противном случае – значение переменной `c`.

Если значение, находящееся в ячейке памяти `0x0b892a98` больше значения, находящегося по адресу `0x0b892a98`, то копирую в ячейку с адресом `0xb892a9c` значение, находящееся по первому адресу, в противном случае – то, что находится по второму адресу.

Типы данных

Вся информация хранится вычислительными машинами в цифровом виде. Независимо от того, какими данными вы манипулируете, будь то числа, текст, музыкальные файлы или видеофайл – все они хранятся в виде последовательности нулей и единиц.
Тем не менее, программисты уже давно работают с данными в их естественном представлении. Т.е. если необходимо вывести строчку, то она вводится не в виде последовательности кодов символов, а в естественном виде – при помощи букв и символов. Все это возможно, благодаря тому, что программист имеет возможность указать, к какому *типу данных* будет относиться та или иная переменная.
В Pascal имеются следующие типы данных:

- целые числа;
- вещественные числа (т.е. те, которые могут иметь дробную часть);
- символы;
- строки;
- логический тип (очень важный, несмотря на то, что может принимать всего два значения: «истина» или «ложь»);
- файловый;
- пользовательский (записи);
- указатели.

Кроме того, некоторые из типов данных (как например целые и вещественные числа) могут содержать в себе различные модификации, отличающиеся друг от друга объемом занимаемой памяти и, следовательно, диапазоном хранимых чисел.
Для того, чтобы понять как зависит диапазон хранимых чисел от объема занимаемой памяти, давайте рассмотрим способы хранения чисел.
Возьмем минимально возможный объем информации – 1 байт. И выясним, сколько чисел и в каком диапазоне в нем можно хранить. Один байт, в свою очередь, состоит и 8 бит, каждый из которых может независимо друг от друга принимать значение либо 0, либо 1. Т.е. все числа хранятся в *двоичном виде*. Например, число 1 представляется следующим образом:

0 0 0 0 0 0 0 1

Число 2 вот так:

0 0 0 0 0 0 1 0

Число 3:

0 0 0 0 0 0 1 1

Число 4:

0 0 0 0 0 1 0 0

и т.д.

Если вспомнить комбинаторику и попытаться посчитать общее количество различных комбинаций нулей и единиц в байте, то мы получим следующее количество: 2^8=256 (две цифры – восемь позиций, следовательно, двойку нужно возвести в восьмую степень). Поэтому один байт может вместить себя не более 256 значений. В зависимости от того, необходимо ли хранить отрицательные значения или нет, диапазон однобайтовых чисел может быть либо от 0 до 255, либо от -128 до 127. Конечно, такого диапазона далеко не всегда хватает, чтобы хранить необходимые для работы программы значения. Поэтому существуют некоторые другие типы данных, занимающие больше места в памяти.

Целые числа

Название типа	Размер	Диапазон допустимых значений
Byte	**1 байт**	**0..255**
ShortInt	**1 байт**	**-128..127**
Integer	**2 байта**	**-32’768..32’767**
Word	**2 байта**	**0..65’535**
LongInt	**4 байта**	**-2’147’483’648..2’147’483’647**

Вещественные числа

Названия типа	Размер	Число значащих цифр	Диапазон допустимых значений
Real	**6 байт**	**11-12**	**$2.9\cdot10^{-39}..1.7\cdot10^{38}$**
Single	**4 байта**	**7-8**	**$1.5\cdot10^{-45}..1.7\cdot10^{38}$**
Double	**8 байт**	**15-16**	**$5.0\cdot10^{-324}..1.7\cdot10^{308}$**
Extended	**10 байт**	**19-20**	**$3.4\cdot10^{-4932}..1.1\cdot10^{4932}$**
Comp	**8 байт**	**19-20**	**$-1.5\cdot10^{-45}..1.7\cdot10^{38}$**

У вещественных (дробных) переменных характеристик на одну больше. Имеет значение, сколько значащих цифр может хранить переменная того или иного типа. Дело в том, что заявленные в диапазоне значения хранятся не полностью. И ведь действительно, ведь например $4.4\cdot10^{22}$ в десятичной системе записывает так:

$$4.4\cdot10^{22}=44000000000000000000000.$$

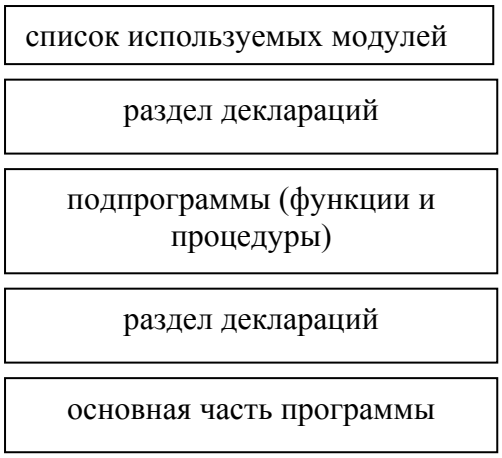
Т.е. видно, что цифр у такого числа довольно много. Поэтому все они не хранятся, а хранится лишь часть из них, наиболее значимая в практических приложениях. Вот количество хранимых цифр и определяет параметр «Число значащих цифр». Чем больше этот параметр, тем точнее хранит тип данных число.

Для хранения вещественного числа, отдельно хранится ее *мантисса* и *экспонента.* Например, число 34567,49324 представляется в виде $3{,}456749324\cdot10^4$. В данном случае, число, которое умножается на десять в какой-то степени, носит название мантиссы, а то, в какую степень нужно возвести число – называется экспонентой. По сути дела – количество чисел, запоминаемых тем или иным типом данных и определяет количество цифр в мантиссе.

Таким образом, если мы хотим сохранить в переменной типа Single число, например, 987654321, то мы сможем (см. таблицу выше) сохранить лишь только 7 значащих чисел, т.е. число будет представлено в виде 9.876543E+0008 (так частенько обозначается запись $9.876543\cdot10^8$, поскольку в текстовом режиме невозможно отображать верхний индекс). В связи с тем, что при использовании разных типов данных одно и то же число может представляться по разному, могут возникать нюансы при использовании вещественных чисел, которые мы рассмотрим более подробно в главе, посвященной *условному оператору*.

Структура программы на Pascal

Теперь у нас уже достаточно знаний, для того чтобы узнать, как выглядит в общем виде любая программа, написанная на Pascal. Каждую логическую часть мы изобразим в виде прямоугольника на схеме ниже.



Все эти составные части программы мы рассмотрим в свое время. На настоящий момент нас интересуют только два блока: раздел деклараций (это где мы объявляем переменные) и раздел, который мы назвали «основная часть программы». Приведем пример программы, состоящих из этих двух частей:

Листинг 3

```
var MyVar, MyVar2, MyVar3: Integer;

begin
  MyVar:=2;
  MyVar2:=3;
  {Результат умножения записываем в переменную MyVar3}
  MyVar3:=MyVar*MyVar2;
end.
```

В приведенном примере разделу деклараций соответствует участок кода (var – это ключевое слово, определяющее начало блока деклараций. Это сокращение от английского слова *variable* – переменная).

```
var MyVar, MyVar2, MyVar3: Integer;
```

А основной части программы – участок между ключевыми словами **begin** и **end.**

```
begin
  MyVar:=2;
  MyVar2:=3;
  {Результат умножения записываем в переменную MyVar3}
  MyVar3:=MyVar*MyVar2;
end.
```

Рассмотрим каждый из этих блоков более подробно.

Раздел деклараций

В этом разделе, как мы уже выяснили выше, находятся описания всех используемых переменных. Описать переменную – это значит дать ей имя и указать к какому типу данных она относится. В общем виде раздел деклараций выглядит следующим образом:

```
var ИМЯ_ПЕРЕМЕННОЙ, …, ИМЯ_ПЕРЕМЕННОЙ: ТИП_ДАННЫХ_1;
    …
    ИМЯ_ПЕРЕМЕННОЙ, …, ИМЯ_ПЕРЕМЕННОЙ: ТИП_ДАННЫХ_N;
```

Таким образом, если необходимо объявить несколько переменных одного типа данных, то их имена можно перечислить через запятую, а после двоеточия, – указать к какому типу данных относится эта переменная. В полном соответствии с этим правилом и выглядит раздел деклараций в Листинг 3. В нем объявлены три переменные с именами `MyVar`, `MyVar2` и `MyVar3`, каждая из которых имеет тип `Integer`. Обратите также внимание на точку с запятой в конце каждого блока объявления переменных. Точка с запятой – это разделитель между конструкциями языка. По этому знаку препинания синтаксический анализатор определяет, где закончился один оператор программы и начался следующий. Приведем еще несколько примеров того, как может выглядеть раздел деклараций:

Листинг 4

```
var a: Integer;
    b: Double;
    c,i:Word;
```

Листинг 5

```
var mm:Integer; dd:Double; flag,ef: Byte;
```

Требования к именам идентификаторов

В Pascal все, чему вы сами придумываете имена (переменные, в дальнейшем мы научимся создавать свои константы, типы данных, процедуры и функции) называется *идентификаторами.* Идентификатор – это уникальное имя, по которому можно однозначно идентифицировать какой-либо объект в создаваемой программе. Это слово выступает синонимом имени. Но к именованию объектов в программе нельзя подходить абы как. Есть правила, которым должны подчиняться все идентификаторы программы. Их не так много:

1. Идентификатор должен состоять только из букв латинского алфавита (больших и маленьких), символа подчеркивания (“_”) и цифр.
2. Идентификатор не должен начинаться с цифры.
3. Не допускается называть одинаково переменные разных типов или два раза объявлять одну и ту же переменную.

Приведем таблицу, в которой приведем примеры правильных идентификаторов и неправильных с пояснением того, почему они являются ошибочными.

Пример декларации	Правильность
`Var F1234567890: Integer;`	**Да, идентификатор верен**
`Var _temp: Double;`	**Да, идентификатор верен**
`Var 1stLine: Byte;`	**Нет, такого идентификатора быть не может, так как его имя не может начинаться с цифры.**
`Var Line counter: Real;`	**Нет, имя переменной содержит недопустимы символ – пробел. Если имя**

	переменной состоит из нескольких слов, то как правило, вместо пробела используется символ подчеркивания: Line_counter.
`Var cOoLVaR: LongInt;`	**Да, это допустимый вариант идентификатора, однако, хаотично перемешивать регистр букв не рекомендуется. Это модно среди молодежи, однако вызывает тошнотворные рефлексы у опытных программистов, поскольку читать исходный код, содержащий такие имена переменных немного затруднительно. Как правило, имя переменной пишется либо целиком «маленькими» буквами, либо выделяется заглавными буквами начало каждого из слов, входящих в название, например: CoolVar.**
`Var d: Double;` `d: Integer;`	**Нет, нельзя давать одинаковые имена нескольким идентификаторам.**

Основная часть программы

Продолжим разбирать пример, приведенный в Листинг 3. Теперь наша задача разобраться в том, что написано между ключевыми словами **`begin`** и **`end`**.
Строчкой

```
MyVar:=2;
```

мы записываем в переменную `MyVar` значение, равное двум. Аналогично, переменной `MyVar2` мы присваиваем значение, равное трем. А в переменную `MyVar3` записываем результат умножения значений, хранящихся в `MyVar` и `MyVar3`.

Никогда не забывайте, что оператор присваивания на языке Pascal записывается при помощи двух символов: двоеточия и знака равенства («:=»).

Обратите также внимание на участок кода, содержащийся в фигурных скобках.

```
{Результат умножения записываем в переменную MyVar3}
```

Это комментарий. В комментариях можно писать все что угодно, т.к. они игнорируются компилятором. Обычно в комментариях пишут сопроводительный текст, по которому легче в дальнейшем разобраться в программе. В Pascal кроме приведенного вида комментариев, есть еще один, который обозначается скобкой со звездочкой:

```
(* Результат умножения записываем в переменную MyVar3 *)
```

Математические операции

Как вы обратили внимание, значения переменных можно не просто задавать в коде программы (например, `MyVar:=2`), но и вычислять. В случае целых и вещественных чисел для этих целей можно использовать различные математические действия

(операции). Некоторые из них обозначаются так же, как и в курсе математики, а для некоторых введены специфичные операторы. Перечислим доступные для программиста математические операции в виде таблицы.

Математические операции в Pascal

Операция	Обозначение в Pascal	Комментарий
Сложение	+	
Вычитание	-	
Умножение	*	
Деление	/	На ноль делить нельзя!
Вычисление остатка от деления	Mod	Вычислять остаток можно только от деления целых чисел. Т.е. этот оператор неприменим к вещественным переменным
Целочисленное деление	Div	Также применим только целочисленным переменным (Integer, LongInt, Word, Byte)

Порядок вычисления выражений определяется приоритетом операций, который во многом схож с приоритетами операций, принятыми в математике. Так, умножение и деление обладают более высоким приоритетом по сравнению со сложением и вычитанием, поэтому результат умножения вычисляется раньше суммы или разности. Но приоритет, по аналогии с математикой, можно задавать при помощи круглых скобок.
Приведем несколько примеров вычисления значения различных выражений. Поскольку в языке Pascal не имеет смысла записывать выражение «просто так», его необходимо куда-либо записать, например, в другую переменную, чтобы впоследствии с этим значением можно было работать. Для всех приведенных ниже примеров, мы будем использовать переменную `result`, считая, что она была объявлена в разделе деклараций соответствующим образом.

Пример 1.

Вычислить значение выражения

$$\frac{a+b}{2}+\frac{a^2}{(a-b)}$$

На Pascal значение этого выражения записывается следующим образом:

```
result:=(a+b)/2 + a*a/(a-b);
```

Пример 2.

Вычислить значение выражения

$$\frac{a\cdot(a+b)-\dfrac{b}{2\cdot a}}{3\cdot b}$$

Чуть-чуть посложнее, но ничего страшного:

```
result:=(a*(a+b)-b/(2*a))/(3*b);
```

Пример 3.

Вычислить остаток от деления 10 на 3

```
result:=10 mod 3;
```

Ввод и вывод значений переменных на экран

Ввод значений

При объявлении переменных значения, хранящиеся в них, вообще говоря, не определены[5]. Значения переменных можно задавать как непосредственно в коде программы, при помощи *оператора присваивания (“:=”)*, так и в ходе выполнения программы. В этом случае значение вводит пользователь, либо оно считывается из файла, либо вообще из каких-либо внешних источников (последовательного порта, сетевого интерфейса и т.д.).
Для того, чтобы пользователь ввел значение с клавиатуры, используются процедуры `ReadLn` (это сокращение от английского выражения read line – прочитать строчку) и `Read` (прочитать). В это процедуре можно указывать неограниченное количество вводимых переменных.
Например:

```
var a, b, c: Integer;
begin
  ReadLn(a, b);
  c:=a+b;
end.
```

В этом примере, пользователя просят ввести значения переменных `a` и `b`, а в переменную `c` записывается результат их сложения.

Вывод значений

Любые данные, которые программа обрабатывает или вычисляет не интересны до тех пор, пока пользователь не может ими воспользоваться. В приведенном выше примере, пользователь вводит два числа для сложения, однако, результат этой операции нигде не отображается. Самым простейшим способом отображение информации является вывод значения переменной на экран. В Pascal это осуществляется при помощи процедуры `WriteLn` (также сокращение от английской фразы Write line – напечатать строчку) и `Write` (напечатать). В скобках, в этой процедуре указываются через запятую переменные, значения которых необходимо вывести на экран.
Модифицируем пример со сложением двух чисел, добавим в него возможность вывода результата на экран.

```
var a, b, c: Integer;
begin
  ReadLn(a, b);
  c:=a+b;
  WriteLn(c);
end.
```

[5] на самом деле Pascal старается обнулять объявляемые переменные.

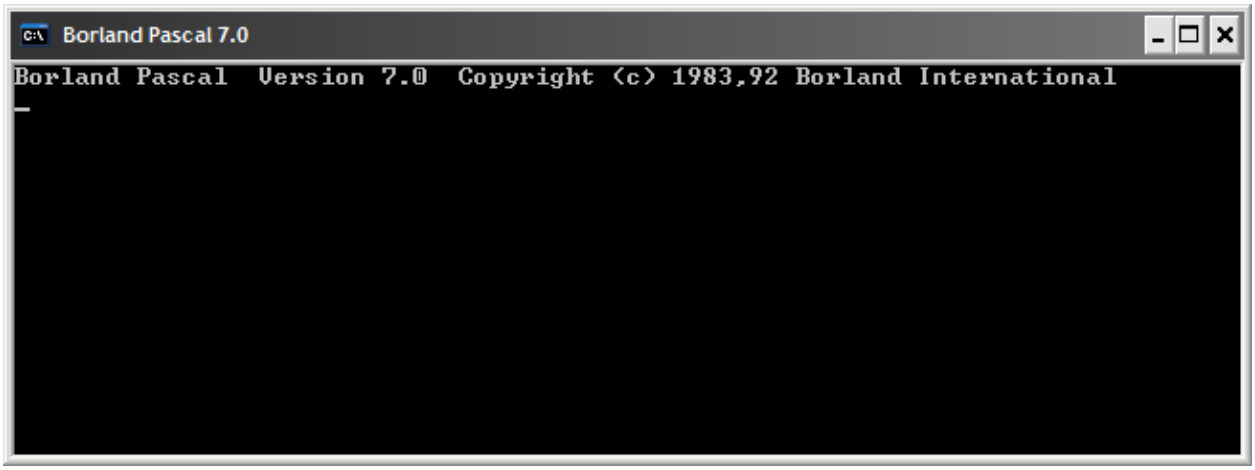
Теперь этот вполне работоспособный код программы, которую можно запустить и проверить ее работоспособность. Для этого, необходимо воспользоваться пунктом меню Run->Run (см.

Рисунок 4) или воспользоваться сочетанием клавиш Ctrl+F9.

Рисунок 4

При этом после запуска программы, пользователь увидит черный экран и мигающий курсор (см. Рисунок 5).

Рисунок 5



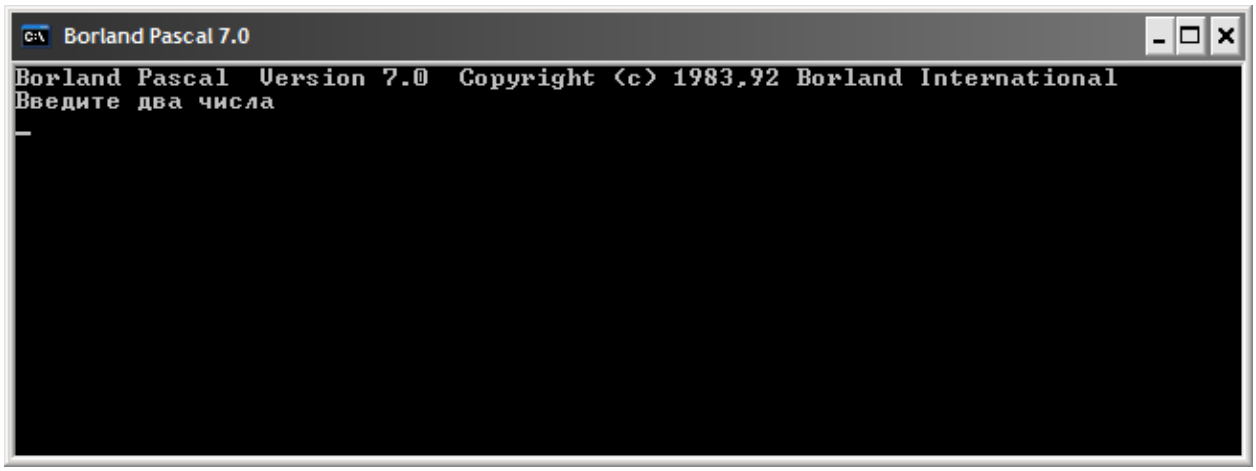
В этом состоянии программа будет находиться до тех пор, пока пользователь не введет два числа. Разумеется, если мы передадим такую программу пользователю, не предупредив о том, что от него требуется, он, возможно, будет долго гадать, что же ему с этой программой делать и, скорее всего, подумает, что программа просто «зависла». Поэтому, что пользователь не гадал, принято информировать его о том, что же от него ожидает программа. В данном случае, перед тем как попросить его ввести два числа, можно вывести поясняющий текст, при помощи той же процедуры `WriteLn`[6]:

```
var a, b, c: Integer;
begin
  WriteLn('Введите два числа:');
  ReadLn(a, b);
  c:=a+b;
  Writeln('Результат a+b=',c);
end.
```

[6] В приведенном примере используются русские буквы для поясняющего текста. Однако, вполне возможно, что в вашей операционной системе русские буквы вводить не получится. Не отчаивайтесь. Можно сохранить исходный текст с русскими буквами при помощи стороннего текстового редактора в кодировке DOS, например UltraEdit, и открыть файл в среде Pascal.

В этом случае, после запуска программы, пользователь увидит строчку текста, заключенную в одинарные кавычки внутри процедуры WriteLn: Введите два числа (см. Рисунок 6).

Рисунок 6



Обратите также внимание, что числа можно вводить построчно (т.е. после каждое числа нажимать кнопку Enter на клавиатуре), так и через пробел. Т.е. в нашем примере можно вводить числа вот так:

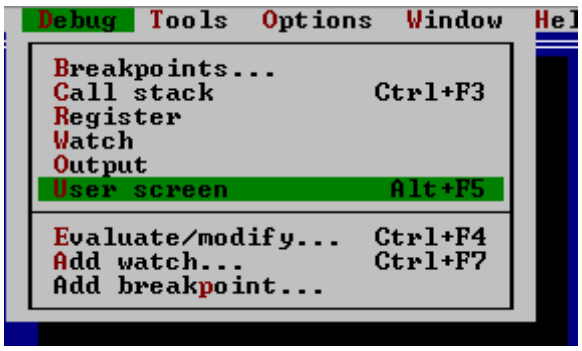
```
2

3
```

и можно вводить числа так:

```
2 3
```

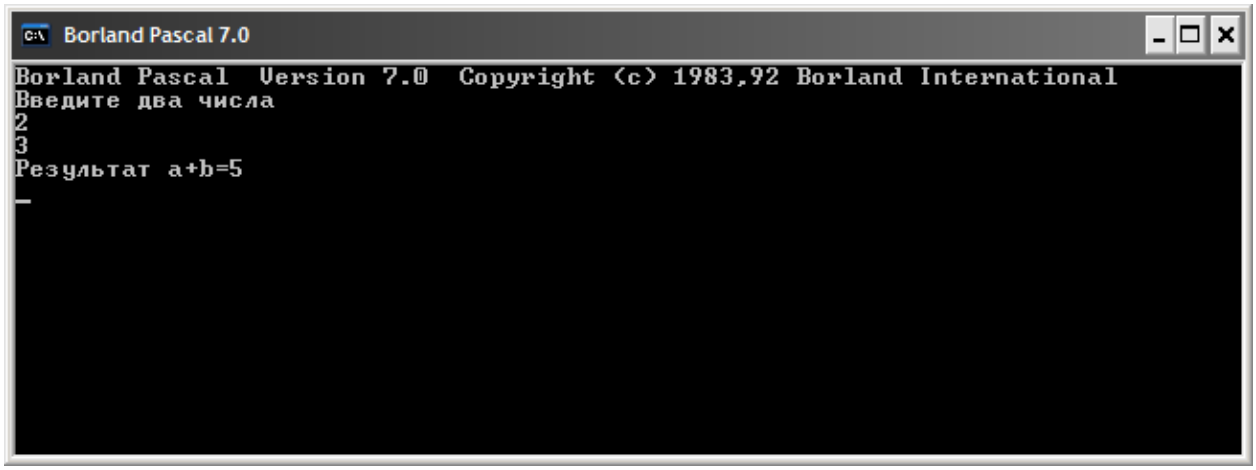
Однако, если вы попробуете запустить программу, то вы заметите, что после того как вы введете два числа, программа мгновенно выполнится и мы не успеем даже посмотреть на результат. Это происходит потому, что после строчки вывода результата на экран (`WriteLn('Результат a+b=', c)`) сразу идет ключевое слово **`end`**, означающее окончание программы. Для того, чтобы все-таки посмотреть результат работы программы, можно в среде Turbo Pascal воспользоваться пунктом меню Debug->Output Screen (см. Рисунок 7) или эквивалентным сочетанием клавиш Alt+F5.

Рисунок 7



Результат должен быть таким, как показано на рисунке ниже.

Рисунок 8



Практикум: Разработка программы «Конвертер единиц измерения»

Необходимо разработать программу, которая переводит введенную пользователем длину в сантиметрах в дюймы и сажени[7].
Для того, чтобы написать такую программу, нам необходимо знать соотношения между сантиметром дюймом и саженью. Всю необходимую информацию можно почерпнуть из Wikipedia:

- 1 см. – это 0,0046869 сажени;
- 1 см. – это 0,3937007 дюйма.

Теперь написать программу ничего не стоит. Нам потребуется три вещественные переменные: первая – для хранения значения, введенного пользователем, вторая – для вычисления значения в дюймах, а третья – для вычисления значения в саженях.

```
var
   cmValue, inchValue, sazhenValue:Real;
Begin
   WriteLn('Введите значение в см.: ');
   ReadLn(cmValue);
   inchValue:=cmValue*0.3937007;
   sazhenValue:=cmValue*0.0046869;
   WriteLn(cmValue, ' см-это ', inchValue, ' дюймов ');
   WriteLn(cmValue, ' см-это ', sazhenValue, ' саженей');
End.
```

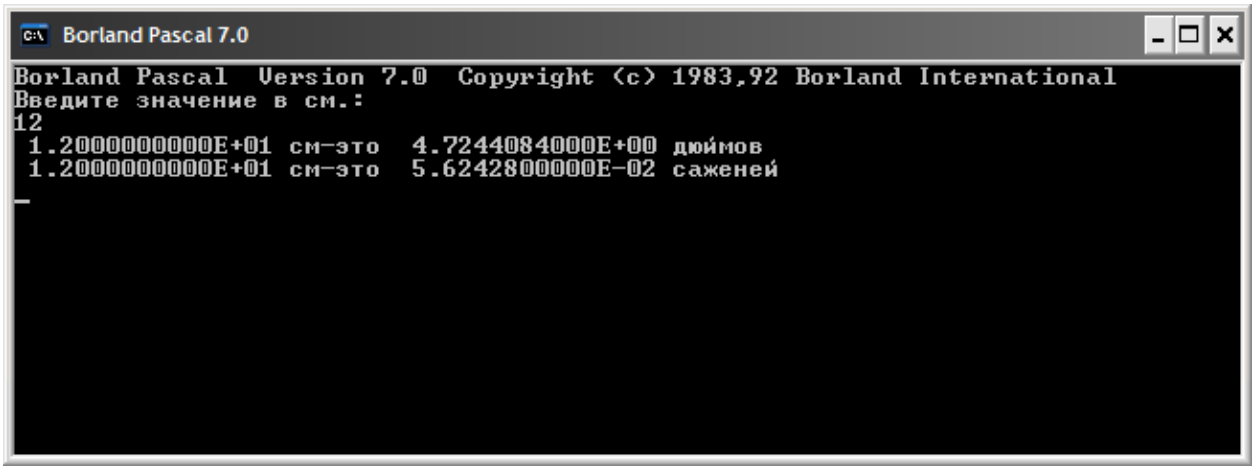
В приведенном примере, можно было бы обойтись и всего одной переменной, ведь вычисленные значения для дюймов и саженей нам больше не потребуются. Тогда и код программы будет значительно короче.

```
var
   cmValue : Real
Begin
   WriteLn('Введите значение в см.:');
```

[7] Сажень – это такая старорусская мера длины. Вышла из употребления в 1924 году.

```
    ReadLn(cmValue);
    WriteLn(cmValue, ' см-это ',cmValue*0.3937007,
            ' дюймов ');
    WriteLn(cmValue, ' см-это ', cmValue*0.0046869,
            ' саженей');
End.
```

Рисунок 9



Практикум: Разработка программы, вычисляющей корни квадратного уравнения

Поставим перед собой задачу написать программу, вычисляющей корни квадратного уравнения[8]. При этом коэффициенты уравнения пользователь должен вводить с клавиатуры после запуска программы.
Напомним, что квадратным уравнением называется уравнения вида:

$$a \cdot x^2 + b \cdot x + c = 0, \quad (1)$$

где $a \neq 0, b, c$ - коэффициенты квадратного уравнения.
Решением этого уравнения являются все такие числа x, при подстановке которых в выражение (1) получается верное равенство.
Из школьного курса известно, что корни квадратного уравнения (а их два) вычисляются по формулам:

$$x_1 = \frac{-b + \sqrt{D}}{2 \cdot a}, \quad (2)$$

$$x_2 = \frac{-b - \sqrt{D}}{2 \cdot a}, \quad (3)$$

где D называется дискриминантом и вычисляется по формуле:

$$D = b^2 - 4 \cdot a \cdot c.$$

Начнем разработку программы с анализа количества необходимых переменных и их типов данных.

[8] Читатель, еще не знакомый с квадратными уравнениями, может пропустить этот раздел, либо прочитать и ознакомиться с квадратными уравнениями самостоятельно.

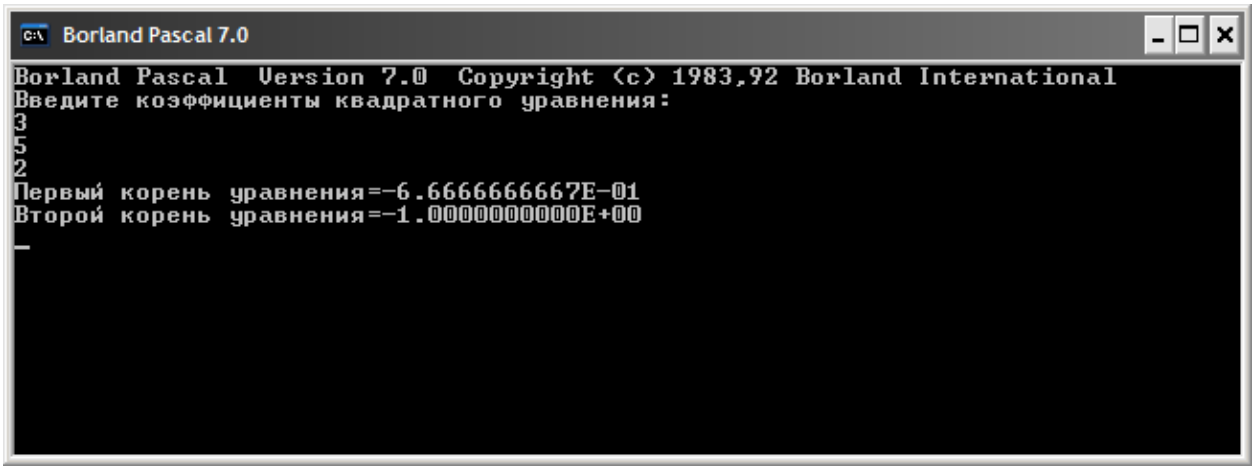
Во-первых, нам потребуются переменные, хранящие значения коэффициентов квадратного уравнения. В общем случае, эти коэффициенты не обязательно будут целыми числами, поэтому вполне логично объявить эти переменные вещественными.
Во-вторых, нам необходимо будет хранить где-то вычисленные корни, следовательно, нам придется объявить еще две переменные, также вещественного типа данных.
В-третьих, мы видим, что в формулах (2) и (3) два раза используется значение квадратного коря из дискриминанта. Наверное, нет смысла его дважды вычислять, поэтому его значение мы вычислим один раз и сохраним в своей, также вещественной, переменной.
Для вычисления квадратного корня воспользуемся стандартной функцией Pascal: `Sqrt`.

Подведем итог: всего нам потребуется объявить 6 вещественных переменных.
Теперь, написание требуемой программы не составит особого труда.

Листинг 6

```
var
  a, b, c   : Real;
  D         : Real;
  x1, x2    : Real;
Begin
  WriteLn('Введите коэффициенты квадратного уравнения: ');
  ReadLn(a, b, c);
  D:= Sqrt(b*b-4*a*c);
  x1:= (-b+D)/(2*a);
  x2:= (-b-D)/(2*a);
  WriteLn('Первый корень уравнения=', x1);
  WriteLn('Второй корень уравнения=', x2);
End.
```

Рисунок 10

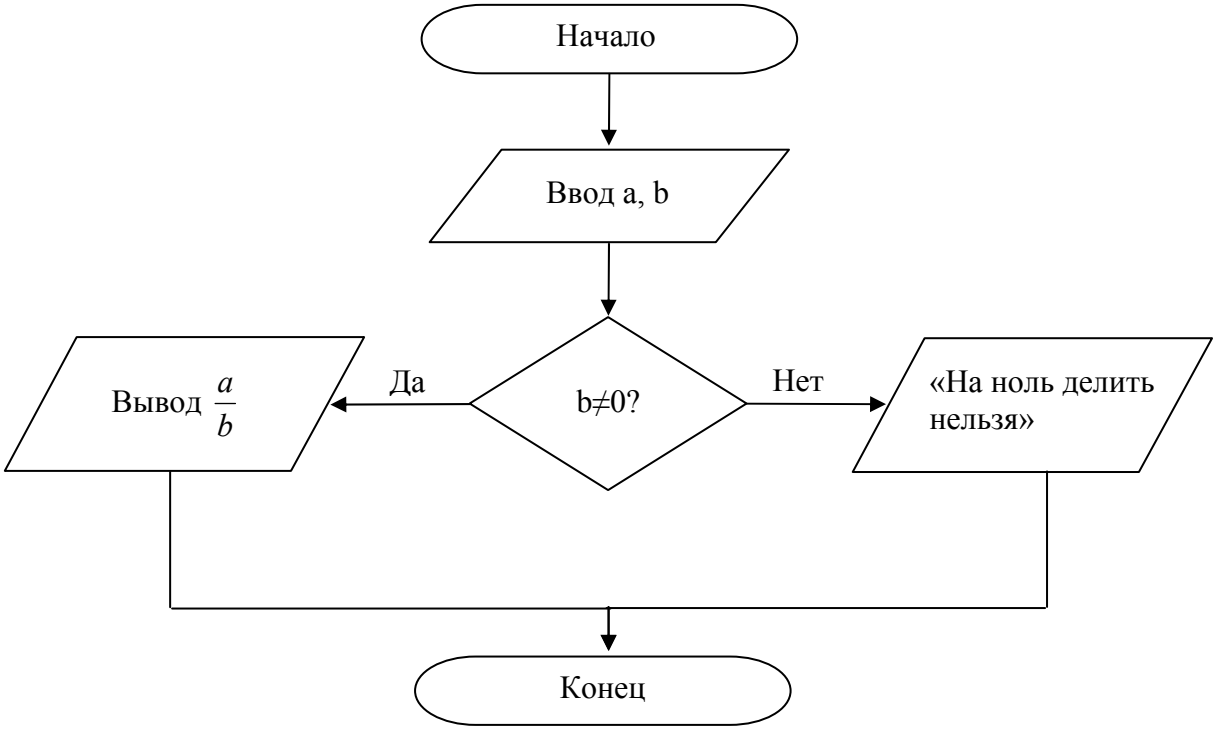


Эта программа будет «вылетать» в случае, если мы введем коэффициенты уравнения, для которого дискриминант будет отрицательным (извлекать квадратный корень из отрицательных чисел мы пока не умеем). Проверять такого рода условия мы научимся в следующей главе при помощи условного оператора.

Модуль 2. Обработка информации и управление ходом выполнения программы

До сих пор мы обладали знаниями, которые могли позволить нам писать программы *линейной структуры*, т.е. которые выполняются *безусловно*: все операторы выполняются сверху вниз и слева направо. Тем не менее, бывают ситуации, когда необходимо один участок кода выполнять в одном случае, а второй – в другом. Например, если мы захотим написать программу, которая делит два числа, то нам необходимо будет проверить на равенство нулю делителя и, в случае, если это так, то вывести предупреждение пользователю, что на ноль делить нельзя. Если изобразить этот процесс графически, то получится следующая схема.

Рисунок 11



Как мы видим, в зависимости от определенного условия может выполняться тот или иной участок кода. Программа приобретает, как говорят, *ветвящуюся структуру.* Но для ее реализации нам необходимо иметь инструменты, позволяющие проверять некоторые условия. Как, например, в данном случае – равенство нулю знаменателя. Для этих целей в языках программирования используется *условный оператор*.

Условный оператор

В общем виде условный оператор в Pascal выглядит следующим образом:

```
If УСЛОВИЕ then
  Begin
       БЛОК_ОПЕРАТОРОВ_1;
  End
Else
  Begin
       БЛОК_ОПЕРАТОРОВ_2;
  End;
```

где

- *УСЛОВИЕ* – это проверяемое логическое условие, которое должно быть либо истинным либо ложным;
- *БЛОК_ОПЕРАТОРОВ_1* – код, который должен быть выполнен, в случае, если *УСЛОВИЕ* истинно*;*
- *БЛОК_ОПЕРАТОРОВ_2* – код, который должен быть выполнен, в противном случае, т.е. если *УСЛОВИЕ* ложно**.**

В качестве примера, реализуем программу, которая графически изображена на схеме выше (см. Рисунок 11).

```
var a, b : Integer;
Begin
WriteLn('Введите делимое и делитель ');
  ReadLn(a, b);
  If b<>0 then
       Begin
            WriteLn('Результат деления = ', a/b);
       End
  Else
       Begin
            WriteLn('Делить на ноль нельзя!');
       End;
End.
```

В данном примере участок кода b<>0 и является проверяемым условием, которое будет истинным, если b отлично от нуля, и ложным – в случае, если пользователь введет нулевое значение.

Обратите особое внимание и никогда не забывайте об этом: перед ключевым словом **Else** условного оператора точка с запятой не ставится никогда!

Кроме того, участок кода, который должен выполняться в случае, если *УСЛОВИЕ* неверно не является обязательным. Иными словами, если в конкретном случае, для реализации логики программы, блок **ELSE** не нужен, то его писать не обязательно. Т.е. в этом случае вид условного оператора может выглядеть следующим образом:

```
If УСЛОВИЕ then
Begin
     БЛОК_ОПЕРАТОРОВ_1;
End;
```

Если БЛОК_ОПЕРАТОРОВ_1 состоит лишь из одного оператора, то *операторные скобки* **Begin** и **End** можно не писать, например

```
If (b=0) then
WriteLn('Делить на ноль нельзя!')
Else
WriteLn('Результат деления', a/b);
```

Поскольку в логических выражениях часто требуется проверять различные равенства и неравенства, то приведем таблицу *операторов сравнения,* используемых в Pascal:

Операция сравнения	Оператор	Пример использования
>	**>**	If **a>b** then …
<	**<**	If **a<b** then …
=	**=**	If **a=b** then …
≤	**<=**	If **a<=b** then …
≥	**>=**	If **a>=b** then …
≠	**<>**	If **a<>b** then …

Логические выражения

В качестве проверяемого условия в операторе **If** могут быть целые *логические выражения*, которые в себе могут сочетать несколько проверяемых параметров. Ведь можно себе представит ситуацию, когда некоторая часть программы должна работать только в случае, если несколько условий выполняются одновременно? Для этого, все эти условия приходится объединять в *логические выражения*.
Например, если мы захотим написать программу-будильник, то нам необходимо будет сравнивать текущее время с заданным, чтобы определить пора уже срабатывать или еще рано, т.е. реализовать следующее логическое условие:

```
Если (текущая секунда = заданной секунде) И (текущая минута =
заданной минуте) И (текущий час = заданному часу) тогда
срабатываем иначе ничего_не_делаем.
```

Как мы видим, несколько условий связываются в одно выражение при помощи связующего слова «И». В Pascal тоже есть такое ключевое слово, которое и означает требование одновременности выполнения условий. Пишется оно «**and**» и называется «логическое И».
Если требуются условия более «мягкие», т.е. чтобы выполнялось хотя бы одно условие в выражении, то, как и в русском языке, используется слово «или». Например,

```
Если (месяц = сентябрь) ИЛИ (месяц = октябрь) ИЛИ (месяц =
ноябрь) тогда на_дворе_осень
```

В Pascal «логическое или» обозначается ключевым словом «**or**».
Разумеется, ключевые слова «И» и «ИЛИ» можно комбинировать, реализуя более замысловатые и сложные выражения.
Запомнить результат действия логических операторов (а так называются ключевые слова **AND** и **OR**) можно, если посмотреть на так называемую таблицу истинности, в которой приведены результаты работы операторов, в зависимости от значения их операндов. Иными словами, каков будет результат действия «**AND**» или «**OR**» в зависимости от того, истины ли условия слева и справа от них. В этой таблице слова TRUE означают «истина», а FALSE – «ложь».

Условие		Результат
A	B	A AND B
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
FALSE	FALSE	FALSE

Как видно из таблицы, результат действия операнда «**AND**» является истиной, только в том случае, когда оба оператора являются истинными.
В случае оператора OR таблица будет выглядеть наоборот:

Условие		Результат
A	B	A OR B
TRUE	TRUE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	FALSE	FALSE

Результат «ложь» может быть только в том случае, если оба операнда – и A и B являются ложными.
Кроме того, как и в обычных математических операциях, в логических операциях тоже есть свои приоритеты. Так, приоритет у оператора AND выше, чем у оператора OR.
Иногда, даже проводят аналогию между арифметическими и логическими операциями. Действительно, если считать, что истинное значение – это единица, а ложное – это ноль, то получим следующую таблицу для операторов AND и OR:

Условие		**Результат**	
A	**B**	**A** AND **B**	**A** OR **B**
1	**1**	**1**	**1**
0	**1**	**0**	**1**
1	**0**	**0**	**1**
0	**0**	**0**	**0**

Глядя на эту таблицу можно заметить некоторое сходство между умножением и логическим «И», а также между сложением и логическим «ИЛИ». Например, если умножить 1 на 1, то получим – 1. Точно также (`TRUE` **`AND`** `TRUE`) =`TRUE`, аналогично – 1+0=1 и (`TRUE` **`OR`** `FALSE`) =`TRUE` и т.д. Благодаря этой аналогии легко запомнить, что приоритет у оператора **`AND`** выше, чем у оператора **`OR`**. Если необходимо приоритет изменить, то можно, так же как и в математических выражениях, использовать круглые скобки.
Еще одним частовстречающимся логическим оператором является *оператор отрицания* **`- Not`**. Это такой оператор, который изменяет значение логического выражения на противоположное. Например, если некоторое логическое условие `A` было истинным, то (**`Not`** `A`) станет ложным и наоборот.
Приведем несколько примеров, в которых требуется построение логических выражений.

Пример 1

Разработать программу, которая по введенному номеру года пользователем определяет високосный ли год или нет.
Небольшая справка: год называется високосным, если его номер делится на 4, не делится на 100, если только не делится на 400, ну а для рядовых программистов это означает увеличение количества дней в году на 1. Таким образом, 2004 год является високосным, потому что его номер делится на 4 и не делится на 100. Но 1900 год не является високосным, поскольку, хоть его номер и делится на 4, он еще делится на 100.

Таким образом, нам необходимо составить такое логическое выражение, которое было бы истинным в случае, если номер года делится на 4 и не делится на 100, либо делится на 400.
Для того, чтобы проверить делится ли какое-нибудь число нацело на другое число, достаточно проверить равенство нулю остатка от деления. Поэтому, при составлении логического выражения воспользуемся оператором **mod.**
Итак, требуемое нам условие следующее:

```
(Year mod 4=0) and (Year mod 100<>0) or (Year mod 400=0)
```

Тогда полностью текст программы, проверяющей високосность года будет следующей:

```
var Year: Integer;
Begin
  WriteLn('Введите номер года: ');
  ReadLn(Year);
If (Year mod 4=0) and (Year mod 100<>0) or (Year mod 400=0)
then
  Begin
       WriteLn('Год високосный');
  End
Else
  Begin
       Writeln('Год невисокосный');
  End;
End.
```

Пример 2

Пользователь вводит свой рост и вес. А программа выносит рекомендации относительного того, избыточен ли вес, либо вес недостаточен.
Программа основывается на известном соотношении между ростом и весом взрослого человека:

$$Рост - 110 \approx Вес$$

Следовательно, если разница в этом выражении больше введенного пользователем веса, то его собственный вес явно недостаточен. В противном случае – избыточен.
Код программы:

```
Var Length, Weight: Integer;
Begin
WriteLn('Введите рост и вес человека ');
ReadLn(Length, Weight);
If (Length-110>Weight) Then
     Begin
          WriteLn('Вес недостаточен! ');
     End
Else
     Begin
          WriteLn('Вес избыточен');
     End;
End.
```

Логический тип данных

Порой бывает необходимо запоминать значение какого-нибудь логического выражения, чтобы несколько раз его не вычислять. Для этого в Pascal предусмотреть особый, логический тип данных – Boolean.
Использование переменных такого типа данных аналогично обычным переменным. В качестве примера использования приведем модифицированную программу нахождения корней квадратного уравнения, в которой уже проверяется условие отрицательности дискриминанта (поскольку нельзя вычислять квадратный корень из отрицательного числа) и равенство нулю коэффициента a перед x^2 (поскольку при этом квадратное уравнение вырождается и превращается в линейное, что приводит к делению на нуль, в случае использования формул (2) и (3)):

```
Var flag:Boolean;
    a, b, c, D : Real;
    x1, x2 : Real;
begin
WriteLn('Введите коэффициенты квадратного уравнения ');
ReadLn(a, b, c);
D:= b*b-4*a*c;
flag:=(D>=0) and (a<>0);
If flag=True then
Begin
     x1:=(-b+sqrt(D))/(2*a);
     x2:=(-b-sqrt(D))/(2*a);
          WriteLn('Первый корень уравнения =', x1);
          WriteLn('Второй корень уравнения =', x2);
End
Else
     WriteLn('Корней нет или уравнение вырожденное');
End.
```

В этом примере, логической переменной `flag` присваивается результат логического выражения `(D>=0) and (a<>0)`, определяющее возможность вычисления корней квадратного уравнения. Если это выражение истинно (а таковым оно будет являться только в случае неотрицательности дискриминанта и неравенства нулю коэффициента `a`), то и значение переменной `flag` также будет истинным, т.е. `TRUE`.
Особо также отметим, что строчка

```
If flag=True then
```

может быть записана в более коротком виде

```
If flag then
```

т.е. проверка на истинность является в Pascal проверкой по-умолчанию (хотя если бы мы хотели проверить значение на ложность, мы бы уже должны были бы записать условие целиком, т.е.: **`If flag=False then`** … или воспользовать оператором Not для отрицания истинности: **`If Not flag then`**)

Вложенный условный оператор

Поскольку блок операторов между **Begin** и **End** условного оператора (как в истинной части, так и в части **ELSE**) может быть произвольный, то допускается вкладывать произвольное количество условных операторов друг в друга (как матрешку), например, так, как показано на приведенном ниже фрагменте кода:

```
If (Condition_one=True) then
Begin
     If (Sub_condition=true) then
          Begin
          …
          End
     Else
          Begin
          …
          End;
End
Else
Begin
     If (Sub_condition_two=true) then
          Begin
               …
          End
     Else
          Begin
          …
          End;
End;
```

Так, в примере 2 мы не учли возможность вывода сообщения о том, что вес нормальный. Модифицируем исходный текст, добавив такую функциональность:

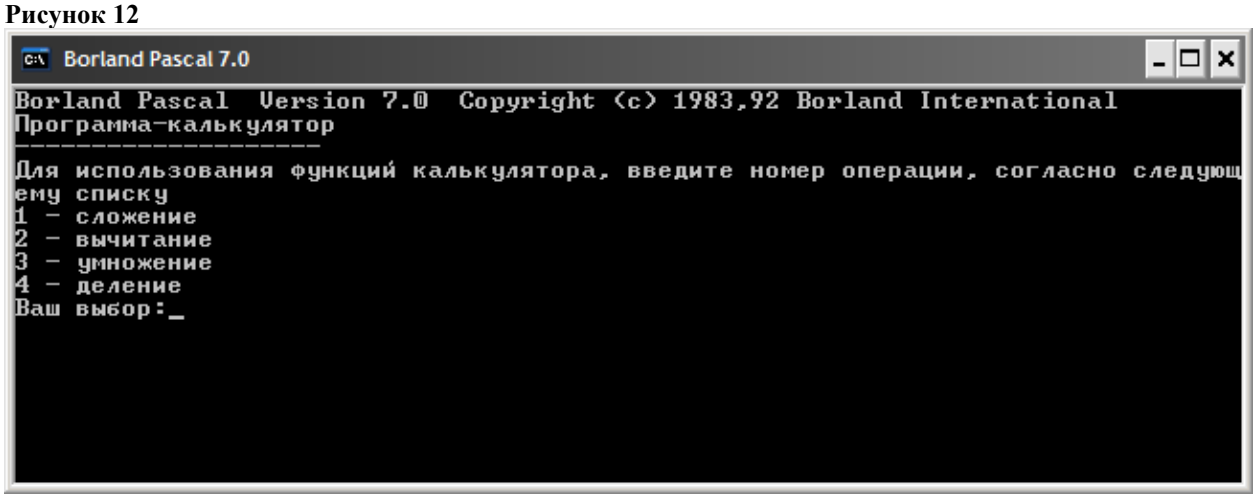
```
Var Length, Weight: Integer;
Begin
WriteLn('Введите рост и вес человека ');
ReadLn(Length, Weight);
If (Length-110>Weight) Then
     WriteLn('Вес недостаточен! ')
Else
     If (Length-110<Weight) Then
          Begin
               WriteLn('Вес избыточен');
          End
     Else
          Begin
               WriteLn('Вес нормальный');
          End;
End.
```

Обратите внимание, что в блоке **Else** внешнего условного оператора, опущены **Begin** и **End.** Это сделано не случайно: дело в том, что весь вложенный **If** рассматривается как один большой оператор, поэтому операторные скобки (**begin** и **end**) можно не писать. По этой же причине опущены операторные скобки в самом первом блоке **If**.

Оператор множественного выбора (case)

В случае, если ветвление более сложное и зависит, например, от значения одного выражения (или переменной), то использование оператора **If** приведет к довольно громоздким конструкциям в исходном тексте программы.
Например, нам необходимо реализовать интерактивное меню, такое, как показано на рисунке ниже:

Рисунок 12



Приведем пример кода, использующий оператор If и реализующий управление ходом выполнения программой в виде такого меню. Т.е. нам нужно, в зависимости от того, какую цифру введет пользователь, выполнять либо сложение, либо вычитание, либо умножении, либо же деление. Предположим, что номер операции записывается в переменную `operation : Byte.`

Листинг 7

```
If operation=1 then
   Begin
{Выполняется сложение}
   End;
If operation=2 then
   Begin
{Выполняется вычитание}
   End;
If operation=3 then
   Begin
{Выполняется умножение}
   End;
If operation=4 then
   Begin
{Выполняется деление}
   End;
```

Если требуется еще реализовать ветвь программы, которая обрабатывает отличный от всех случай (т.е. операция не равна ни 1, ни 2, ни 3 и ни 4), то структура будет и того сложнее и запутанней.
Как раз для таких случаев был введен оператор множественного выбора – **case**. В общем виде его структура выглядит следующим образом:

```
Case ВЫРАЖЕНИЕ Of
value1 : Begin
           БЛОК_ОПЕРАТОРОВ_1;
     End;
value1 : Begin
           БЛОК_ОПЕРАТОРОВ_1;
     End;
…
valueN : Begin
           БЛОК_ОПЕРАТОРОВ_N;
     End;
Else
Begin
           БЛОК_ОПЕРАТОРОВ;
End;
End;
```

Здесь, в качестве ВЫРАЖЕНИЯ может быть использовано любое выражение, которое приводит либо к целочисленному результату, либо к символьному (т.е. результат – один символ[9]). Недопустимо использовать выражения или переменные, которые, например, возвращают целую строку или вещественное число.
value1, value2, …, valueN – это значения, которые может принимать ВЫРАЖЕНИЕ;
БЛОК_ОПЕРАТОРОВ_1, БЛОК_ОПЕРАТОРОВ_2, …, БЛОК_ОПЕРАТОРОВ_N – это операторы, которые необходимо выполнять в зависимости от значения ВЫРАЖЕНИЯ.
Блок Else здесь используется в том же самом смысле, что и в условном операторе. Он предваряет блок операторов, который необходимо выполнить, если среди перечисленных значений value1, …, valueN не нашлось такого, которому бы равнялось ВЫРАЖЕНИЕ.
Используя оператор множественного выбора, программа, приведенная в Листинг 7, будет выглядеть компактнее и понятнее:

```
Case operation Of
1: Begin
     {выполняется сложение}
   End;
2: Begin
     {выполняется вычитание}
   End;
3: Begin
     {выполняется умножение}
   End;
4: Begin
     {выполняется деление}
   End;
End;
```

Еще одно преимущество оператора множественного выбора заключается в том, что вместо одиночного значения ВЫРАЖЕНИЯ можно использовать несколько, а также целые диапазоны.

Пример 1

[9] О символьном типе данных (char) мы поговорим позднее.

В этом примере мы покажем, как можно указывать несколько значений ВЫРАЖЕНИЯ. Как видно, каждое из значений, для которого должен выполняться один и тот же блок операторов, записываются через запятую.

```
Case number Of
1,3,5,7,11,13,17,19: WriteLn('Число ', number,
                                ' – простое');
2,4,6,8,10,12,14,16,18,20: WriteLn('Число ', number,
                                   ' – четное');
Else
     WriteLn('Число ', number,
             ' – нечетное и не является простым');
End;
```

Пример 2

В этом примере мы рассмотрим способ указания целого диапазона, в который может попадать значение ВЫРАЖЕНИЯ оператора множественного выбора. Диапазон значений указывается начальным и конечным значениями, разделенными двумя точками (без всяких пробелов):

```
Case age Of
   0..6: WriteLn('Учащийся детского сада');
  7..16: WriteLn('Учащийся школы');
 17..22: WriteLn('Учащийся МГТУ им. Н. Э. Баумана');
Else
WriteLn('Уже не учащийся');
End;
```

Практикум: Разработка программы, проверяющей возможность существования треугольника с заданными сторонами

Как известно, у треугольника стороны могут быть не всякими. Например, не может быть треугольника со сторонами 1м. , 1м. и 100 м. Предположим, что пользователь вводит три числа, и наша задача выяснить может ли существовать треугольник с такими сторонами. Для этого воспользуемся неравенством, которое в школе получило название *неравенства треугольника:*
для всех сторона *a, b, c* треугольника должны одновременно выполняться неравенства:

$$\begin{cases} a+b>c \\ b+c>a \\ a+c>b \end{cases}.$$

Раз условия должны выполняться одновременно, то это сразу наводит на мысль об использовании логического оператора **AND**, который в данном случае будет связывать три условия, которые представляют запись каждого из этих неравенств.
Для того, чтобы пользователь мог ввести три стороны треугольника, объявим три переменные вещественного типа.
Код такой программы несложен и выглядит следующим образом:

```
Var a, b, c : Real;
```

```
Begin
  WriteLn('Введите три стороны');
  ReadLn(a, b, c);
  If (a+b>c) and (b+c>a) and (a+c>b) then
      WriteLn('Такой треугольник существовать может')
Else
      Writeln('Такого треугольника существовать не может!');
End.
```

Практикум: Вывод названия года в старояпонском календаре по номеру года

Известно, что в старояпонском календаре был принят двенадцатилетний цикл, при этом каждый год имел свое название. Началом цикла будем считать 1996 год – год крысы. Поскольку цикл регулярный, то для того, чтобы вывести его название достаточно анализировать остаток отделения номера года на 12. Осталось только выяснить правильный порядок названий, который опять-таки можно почерпнуть в Wikipedia: крыса, корова, тигр, заяц, дракон, змея, лошадь, овца, обезьяна, петух, собака и свинья.
В этом случае, исходный текст программы выглядит следующим образом:

```
Var Year : Integer;
Begin
WriteLn('Введите номер года ');
ReadLn(Year);
   Case (Year mod 12) Of
    0: WriteLn('Год Обезьяны');
    1: WriteLn('Год Петуха');
    2: WriteLn('Год Собаки');
    3: WriteLn('Год Свиньи');
    4: WriteLn('Год Крысы');
    5: WriteLn('Год Коровы');
    6: WriteLn('Год Тигра');
    7: WriteLn('Год Зайца');
    8: WriteLn('Год Дракона');
    9: WriteLn('Год Змеи');
   10: WriteLn('Год Лошади');
   11: WriteLn('Год Овцы')
  End;
End.
```

Дополнительный материал к модулю 1 и к модулю 2

В этой главе описываются некоторые аспекты языка Pascal, которые не вошли в предыдущие главы, но которые знать нужно для понимания следующих глав.

Подключение дополнительных модулей

В процессе разработки программ, вы в скором времени заметите, что некоторые однотипные действия необходимо выполнять довольно часто. Разумеется, каждый раз писать код, реализующий эти действия не эффективно. Поэтому во многих языках программирования, в том числе и Pascal, можно выносить часть функционала в своеобразные библиотеки – отдельные модули. В программе, в которой планируется использовать эту библиотеку, необходимо будет указать имя модуля и в нужном месте программы осуществить вызов нужного функционала.
В Pascal подключение модуля (т.е. сообщение компилятору о том, что мы используем еще и модуль в своей программе) осуществляется при помощи ключевого слова `uses`, которое надо помещать в самом начале исходного текста программы, еще до раздела деклараций:

```
uses ИМЯ_МОДУЛЯ1, ИМЯ_МОДУЛЯ2, … ИМЯ_МОДУЛЯ2;
```

Например, довольно часто нам потребуется подключать модуль, под названием CRT:

```
uses crt;
```

Приведем несколько примеров программ, которые используют функционал различных модулей, написанных другими программистами.

Пример 1. Модуль CRT - очистка содержимого экрана

Как вы заметили, результаты нескольких запусков программы никуда не исчезают, а остаются на экране (см. Рисунок 13).

Рисунок 13

поэтому хотелось бы научиться очищать, то, что осталось после предыдущих запусков приложений.
Для этих целей предназначена процедура ClrScr модуля Crt. Ее можно использовать в любой части программы, по мере необходимости убирая ненужный текст, оставшийся после работы программы.

Как правило, почти в любой программе перед началом ее работы необходимо очистить экран. Приведем пример того, как это делается на примере программы, проверяющей возможность существования треугольника с заданными сторонами.

```
Uses CRT;    {Подключаем модуль CRT}
Var a, b, c : Real;
Begin
  ClrScr;    {Очищаем экран}
  WriteLn('Введите три стороны');
  ReadLn(a, b, c);
  If (a+b>c) and (b+c>a) and (a+c>b) then
      WriteLn('Такой треугольник существовать может')
Else
      Writeln('Такого треугольника существовать не может!');
End.
```

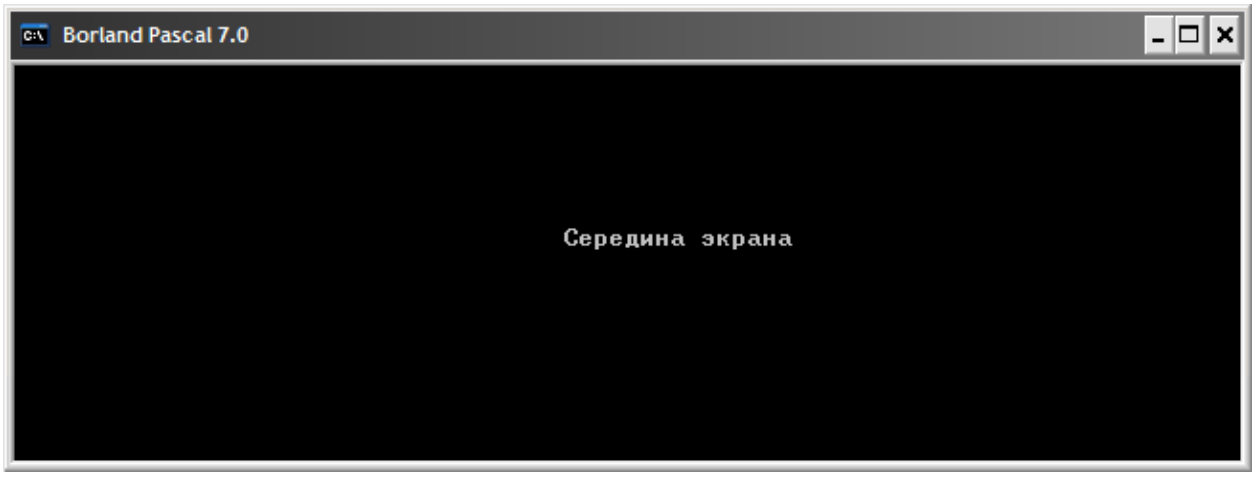
Пример 2. Модуль CRT – Работа с текстом

Модуль CRT также предоставляет процедуры и функции, которые делают работу выводом текста на экран более удобной.

Позиционирование курсора

По-умолчанию, программа выводит текст слева направо и сверху вниз. Однако, в некоторых случаях необходимо размещать текст в строго определенном месте экрана, указывая логические координаты первого символа текста.
Экран имеет строго определенные размеры, равные 80 символов в ширину и 25 символов в высоту (или – 25 строчек и 80 колонок). Исходя из этих размеров можно, например, поместить текст посередине экрана, на 12 строчке и начиная, скажем, с 40-го символа. Для того, чтобы поместить курсор в точку с этими координатами используется процедура, объявленная в модуле CRT: `GotoXY(Координата X,Координата Y)`. Разумеется, позиционирование необходимо осуществлять до того как текст выводится на экран, т.е. сначала вызывать процедуру `GotoXY`, а уж потом использовать `Write` или `WriteLn`.

```
Uses crt;
Begin
  ClrScr;
  GotoXY(40,25);
  WriteLn('Середина экрана');
End.
```



Изменение цвета текста

В модуле CRT также содержатся средства, которые позволяют изменить цвет выводимого текста – это процедура `SetTextColor`. В качестве ее аргумента указывается номер цвета. Цветов всего 16: от 0 до 15. Вместо числовых значений можно использовать мнемонические:

Мнемоническое название	Значение	Цвет
Black	0	Черный
Blue	1	Синий
Green	2	Зеленый
Cyan	3	Ярко-голубой (циан)
Red	4	Красный
Magenta	5	Сиреневый (мажента)
Brown	6	Коричневый
LightGray	7	Светло-серый
DarkGray	8	Темно-серый
LightBlue	9	Светло-голубой
LightGreen	10	Светло-зеленый
LightCyan	11	Светлый циан
LightRed	12	Светло-красный
LightMagenta	13	Светлая мажента
Yellow	14	Желтый
White	15	Белый

Например, чтобы вывести светло-зеленую надпись, необходимо написать следующий код:

```
Uses CRT;
Begin
  ClrScr;
  TextColor(LightGreen);
  WriteLn('Зеленая надпись');
End.
```

Пример 3. Модуль DOS - Получение текущей даты и времени

Существует большой класс программ, для функционирования которых необходимо получать системную дату и/или время. Это как всевозможные будильники, «напоминалки», так и планировщики заданий, выполняющие по расписанию разнообразные сервисные функции. Для реализации задуманного функционала можно использовать процедуры из модуля DOS.

Процедура модуля DOS, возвращающая текущую системную дату, требует указать для нее четыре переменные для года, месяца, дня и номера дня недели, причем каждая из этих переменных должна иметь тип данных `Word`. Называется эта процедура `getDate`:

```
uses Dos;
Var Year, Month, Day, DayOfWeek:Word;
Begin
  getDate(Year, Month, Day, DayOfWeek);
  WriteLn('Сегодня ', Day, '.', Month, '.', Year);
end.
```

Для процедуры, получающей текущее время, потребуется также четыре переменные: для текущего часа, текущей минуты, текущей секунды и для сотой доли секунды. Процедура называется аналогично: getTime.

```
uses Dos;
Var Hour, Minute, Second, HSecond:Word;
Begin
  getTime(Hour, Minute, Second, HSecond);
  WriteLn('Сейчас ', Hour, ':', Minute, ':', Second);
end.
```

Форматированный вывод

В приведенных выше примерах, особенно в тех, что использовали вещественные типы данных, не уделялось внимание «красоте» выводимых значений на экран. А ведь от того насколько пользователю легко будет прочитать результаты работы программы будет зависеть удобство использования. Однако, если вещественные числа выводятся в формате `2.3550000000E+01`, то рядовому пользователю будет неудобно адекватно воспринимать результаты работы программы. Поэтому в этом пункте мы расскажем о том, как можно выводит числовые результаты красиво, превращая такого рода числа в удобочитаемый вид.

Целые числа

В случае выводе целочисленных переменных (Byte, Word, Integer, LongInt), несмотря на то, что у них отсутствует дробная часть, можно указывать сколько символов будет занимать число на экране. Это необходимо в тех случаях, когда, например различные числа требуется выводить в виде таблицы.
Например, если мы выполним следующий код

```
Uses crt;
Begin
  ClrScr;
  WriteLn('Номер | Количество');
  WriteLn(2,3);
  WriteLn(5,6);
  WriteLn(123,443);
  WriteLn(12442,3);
  WriteLn(33,13455);
End.
```

Мы получим следующий результат:

Рисунок 14

На этом рисунке видно, что цифры слиплись и их необходимо разделять пробелами, чтобы пользователь мог понять, где кончается одно число и начинается другое. Однако, поскольку количество разрядов разное (в некоторых столбцах числа состоят из одной цифры, в некоторых, например, из пяти), то определение количества пробелов, которые необходимо добавлять – дело весьма хлопотное.
В таких случаях указывается максимальная длина поля, которое может занимать число. Осуществляется это непосредственно в процедуре `WriteLn` при помощи двоеточия «`:`», которое ставится сразу после числа, либо после переменной. Например, если некоторое число `Number` должно занимать на экране 5 символов, то пишут следующим образом:

```
WriteLn(Number:5);
```

Если числу требуется меньше места, чем заявлено в процедуре `WriteLn`, то оставшиеся неизрасходованные позиции будут заполнены пробелами.
Например, если используется следующая запись

```
WriteLn(12:5);
```

То это означает, что Pascal выделит на экране место из пяти символов и заполнит их согласно схеме, приведенной ниже

_	_	_	**1**	**2**

Если для вывода числа требуется больше символов, чем декларируется, то размер поля автоматически будет увеличен.
Используя такой способ форматирования вывода целочисленных переменных модифицируем предыдущий код, чтобы цифры выводились в удобочитаемой форме:

```
Uses crt;
Begin
  ClrScr;
  WriteLn("Номер | Количество");
  WriteLn(2:5,3:12);
  WriteLn(5:5,6:12);
  WriteLn(123:5,443:12);
  WriteLn(12442:5,3:12);
  WriteLn(33:5,13455:12);
```

```
End.
```

Результат работы такой программы приведен на рисунке ниже

Рисунок 15

Вещественные числа

В случае вещественных чисел мы можем не только указывать количество символов, которое необходимо выделить на отображение числа, но и точность, или, иными словами – количество знаков после запятой. Реализуется аналогичным с целочисленными переменными способом.

```
WriteLn(RealNumber:Длина:Точность);
```

Например, в результате выполнения следующего кода

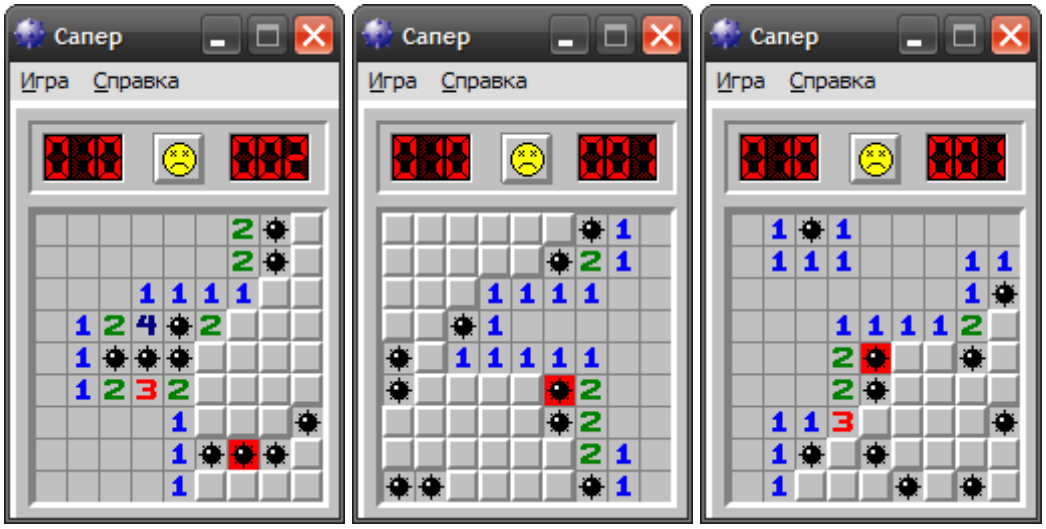
```
Uses crt;
Var f: Real;
Begin
  ClrScr;
  f:=3.14159;
  WriteLn(f:6:2);
End.
```

будет выведено на экран «␣␣3.14», т.к. на все число (включая десятичную точку) было выделено 6 символов, а на часть после запятой – 2.

Генерация псевдослучайных значений

Во многих приложениях, начиная от игр и заканчивая серьезными криптографическими системами, существует потребность в генерации последовательности случайных чисел. Самый просто пример – это программа «Сапер», поставляемая со всеми операционными системами семейства Windows. В этой программе мины каждый раз располагаются в совершенно случайных местах (на приведенном ниже примере мы специально проиграли, чтобы продемонстрировать случайное расположение мин).

Рисунок 16



Конечно, ни один алгоритм не может выдавать в качестве результата абсолютно случайные числа. Это бы противоречило бы самому определению алгоритма как четко определенной последовательности действий[10]. Поэтому используются алгоритмы, которые формируют *почти* независимые друг от друга числа.

Чтобы генерировать псевдослучайное число в Pascal необходимо выполнить два шага:

1. инициализировать генератор псевдослучайных чисел при помощи процедуры `randomize;`
2. получить сгенерированное число, используя функцию `random()`.

Инициализация (или, грубо говоря – «включение и подготовка к работе») необходима для того, чтобы генератор случайных чисел не генерировал повторяющиеся значения. Инициализацию достаточно выполнить один раз за время работы программы, например – в самом ее начале.

Функция `random()` генерирует либо вещественные числа от 0 до 1, либо целые числа в диапазоне от 0 до числа, указанного в качестве ее параметра (в скобках).

Приведем пример использования генератора псевдослучайных чисел в Pascal:

```
Uses crt;
Var intRnd: Integer;
  floatRnd: Real;
Begin
  ClrScr;
  Randomize; {Инициализация генератора}
  intRnd:=random(10); {в переменную intRnd будет записано
                       целое случайное число от 0 до 10}
  floatRnd:=random; {в переменную floatRnd будет записано
                     вещественное случайное число от 0 до 1}
  WriteLn('Случайное целое число: ', intRnd);
  WriteLn('Случайное вещественное число: ', floatRnd:6:2);
End.
```

[10] Правда, существуют специальные Web-сервисы, которые возвращают действительно случайные числа. Но это уже программно-аппаратная платформа, представляющая собой радиоприемник, настроенный на специфическую, неиспользуемую людьми волну, оцифровывающей сигнал и транслируя его. Кроме того, существуют действительно генераторы случайных чисел, например устройство /dev/random в операционных системах LINUX\UNIX, считающее такты процессора, но и они обладают некоторыми недостатками.

Модуль 3. Операторы повторения (циклы)

Представим себе ситуацию, что нам требуется написать программу, которой необходимо вычислить сумму 100 чисел, которые подряд, одно за другим, вводит пользователь. В таком случае, обладая теми знаниями, которые мы получили ранее, нам придется написать программу, вроде той, что приведена ниже

```
Uses Crt;
Var currentNumber, summValue: Integer;
Begin
  summValue:=0;
  WriteLn('Введите число с номером 1');
  ReadLn(currentNumber);
  summValue:=summValue+currentNumber;
  Writeln('Введите число с номером 2');
  ReadLn(currentNumber);
  summValue:=summValue+currentNumber;
  Writeln('Ведите число с номером 3');
  ReadLn(currentNumber);
  summValue:=summValue+currentNumber;
  …
  WriteLn('Введите число с номером 100');
  ReadLn(currentNumber);
  summValue:=summValue+currentNumber;
  WriteLn('сумма 100 чисел равна: ',summValue);
End.
```

Мы поставили многоточие, чтобы не писать еще 96 раз вывод приглашения, ввод значения переменной и суммирование. Конечно, писать такие программы очень неудобно. Поскольку, как видно из примера, исходный текст может очень сильно разрастаться за счет повторяющихся элементов, при этом затрудняется его отладка (представьте, что вы в какой-нибудь строчке ошиблись…). Поэтому, для многократного выполнения некоторых участков кода используются *операторы повторения* или *циклы*.

В зависимости от конкретной задачи, могут быть использованы различные виды циклов:

- циклы с параметром;
- циклы с условием;
  - с предусловием;
  - с постусловием.

Цикл с параметром

Такие циклы применяются в тех случаях, когда точно известно количество итераций (т.е. повторений), которое необходимо произвести, либо это количество можно вычислить. Совершенные повторения считаются в специальной целочисленной переменной, которая называется *параметром цикла* (или *переменной цикла*), а сам цикл, благодаря такой переменной получил название *циклом с параметром.*

Для того, чтобы указать корректно указать, количество итераций такого цикла, необходимо указать начальное значение параметра и его конечно значение.

Синтаксис выглядит следующим образом:

```
For параметр_цикла:=начальное_значение To конечное_значение Do
Begin
   {тело цикла}
End;
```

В данном примере, *{тело цикла}* – это тот участок кода, который необходимо *итерировать,* т.е. повторять. При этом, внутрь тела цикла можно внедрять значение параметра цикла, тем самым, внося изменяющуюся в зависимости от номера итерации цикла переменную в код.
Приведем простой пример, в котором на экран 10 раз будет выводиться строка, содержащая свой номер:

```
Uses Crt;
Var i: Integer;
Begin
ClrScr;
  For i:=1 To 10 Do
  Begin
       WriteLn('Это строка номер ', i);
  End;
End.
```

Результат работы такой программы будет выглядеть следующим образом:

Рисунок 17

В данном примере, в качестве параметре цикла была задействована переменная `i`, изменяющаяся от 1 до 10. Кроме того, ее значение было использовано в процедуре `WriteLn` для того, чтобы определить номер итерации, которая по логике работы программы совпадает с номером выводимой строчки.
В параметрических циклах начальное значение параметра должно быть всегда меньше или равно конечному значению параметра. Иными словами, нельзя организовывать следующий цикл:

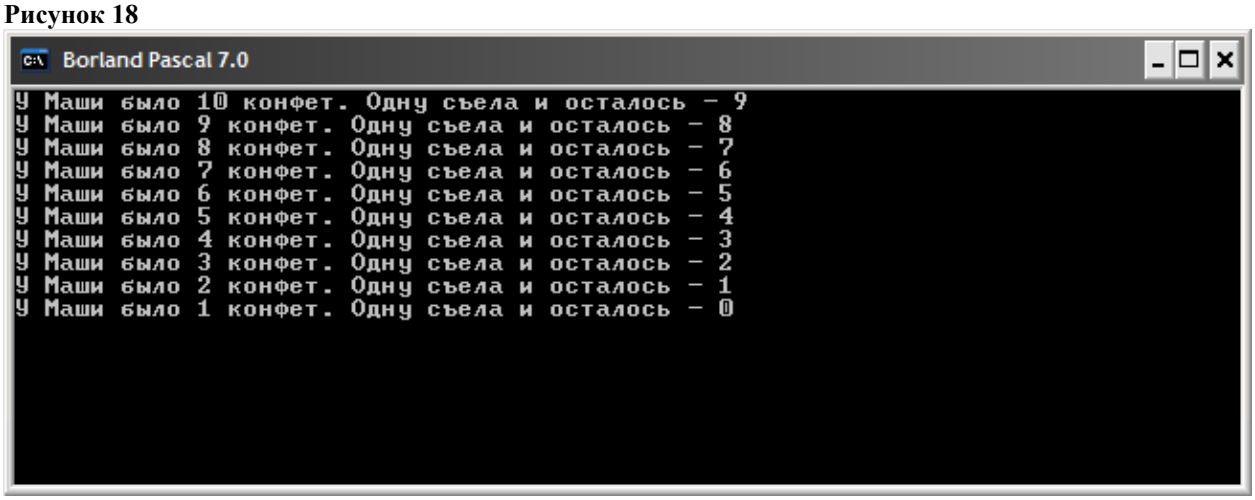
```
For i:=10 To 0 Do {так делать нельзя!}
…
```

Если все-таки необходимо организовать изменение параметра от большего значения к меньшему, то вместо ключевого слова **To** используется ключевое слово **DownTo**.
Например:

```
Uses Crt;
Var i: Integer;
Begin
ClrScr;
```

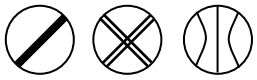
```
For i:=10 DownTo 1 Do
Begin
  WriteLn('У Маши было ', i, ' конфет. Одну съела и',
  ' осталось - ', i-1);
End;
End.
```

Рисунок 18



Практикум: Вычисление факториала числа

Применим цикл с параметром для решения задачи о перестановках.
Суть задачи сводится к подсчету количества различных перестановок, которые можно осуществить из N предметов.
Предположим, что у нас есть три различных мячика, например таких, которые изображены ниже на рисунке



Вопрос заключается в том, сколько существует неповторяющихся вариантов их взаимного расположения?
Для ответа на этот вопрос переберем все возможные варианты расположения мячиков. В результате чего увидим, что всего таких вариантов может быть 6.

1			
2			
3			
4			
5			
6			

Конечно, в общем случае произвольного N, перебирать все возможные варианты не очень эффективно, поэтому необходимо разработать алгоритм подсчета, возвращающий количество неповторяющихся перестановок в зависимости от числа предметов – N. В курсе математики показывается, что количество C таких перестановок вычисляется следующей формулой:

$$C = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (N-1) \cdot N = N!,$$

т.е. это произведение всех чисел от 1 до N. Такое произведение в математике называется *факториалом* числа N. Действительно, если $N = 3$, то $N! = 3! = 1 \cdot 2 \cdot 3 = 6$, что и показал наш прямой перебор всех вариантов.

Факториал – это очень быстрорастущая функция. Поэтому для ее вычисления нам потребуется тип данных, способный хранить числа как можно большие по своему значению. Мы будем использовать тип `LongInt`, хотя и этот тип данных не обеспечит вычисления факториала чисел, больших 12. Иногда применяют для вычисления таких больших значений вещественные числа, выводя потом результат с некоторой точностью.

Для вычисления факториала воспользуемся следующим соображением:

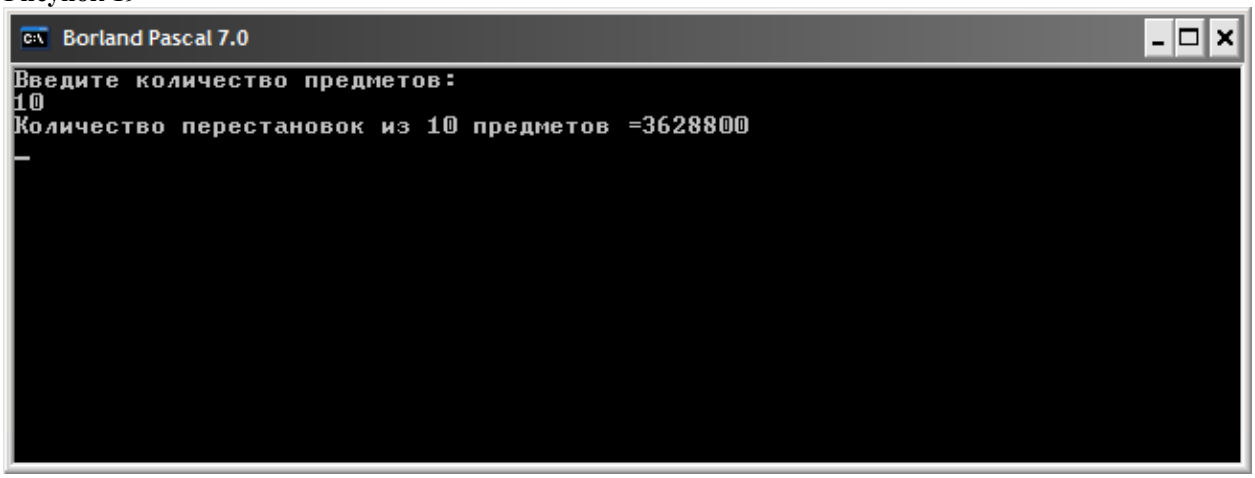
$$N! = \underbrace{1 \cdot 2 \cdot \ldots (N-1)}_{(N-1)!} \cdot N$$

т.е. для того чтобы вычислить факториал числа N, достаточно на N умножить факториал числа, меньшего на единицу. При этом считается, что 1!=1.

Тогда, если мы некоторой переменной в начальный момент присвоим значение, равное 1, а потом будем постепенно умножать значение переменной на параметр цикла, то в итоге мы и получим требуемый результат – факториал числа N. Для этого параметр цикла должен будет перебрать все множители – от 1 до N:

```
Uses Crt;
Var factorial : LongInt;
  i, n: Integer;
Begin
ClrScr;
WriteLn('Введите количество предметов: ');
ReadLn(n);
factorial:=1;
For i:=1 To n Do
  factorial:=factorial*i;
WriteLn('Количество перестановок из ',n, ' предметов =',
factorial);
End.
```

Рисунок 19



Обратите внимание, что в случае использования циклов ситуация с операторными скобками `Begin … End` точно такая же, что и в случае условного оператора: если тело цикла представляет собой всего один оператор (в данном случае – присвоение переменной `factorial` результата умножения), то операторные скобки можно не писать.

Контроль арифметического переполнения

В приведенном выше коде программы таится очень коварная ошибка. Если ввести в качестве количества предметов число, превышающее 12, например 17, то программа выдаст следующий результат:

```
Количество перестановок из 17 предметов =-288522240
```

В то время как, для количества предметов 12:

```
Количество перестановок из 12 предметов =479001600
```

Видно, что количество перестановок из 12 предметов получилось больше чем из 17, не говоря уж о том, что значение факториала не может быть отрицательным!
Таким образом, программа выдает совершенно неправильный результат в некоторых случаях и никак на это не реагирует. Хорошо, это просто случай и его можно легко обнаружить, проанализировав результат. Однако в некоторых случаях, зависимость результата от входных данных программы менее очевидная, однако программа должна гарантировать верность всех выдаваемых результатов.
Давайте для начала разберемся, почему же возникла такая ситуация?
Тип `LongInt`, используемый нами для вычисления факториала, может хранить максимальное число, равное 2147483647. Однако, число 17! превышает это максимально допустимое значение, т.е., грубо говоря, не вмещается в рамки памяти, выделяемые под результат (`LongInt` занимает в памяти 4 байта). Такая ситуация носит название в программировании *переполнением (overflow)*. Для того, чтобы отловить такие исключительные ситуации, в Pascal можно использовать специальные директивы компилятора, определяющие поведение программы при различных типах непредвиденных программистом случаях. Директивы представляют собой комментарий специального вида, размещаемые, как правило, в самом начале кода (но есть и исключения), содержащие знак доллара ($), литеру директивы (для каждого случая предусмотрена своя буква) и знак плюс или минус, обозначающие включение или выключение соответствующего режима компиляции.
Так, для включения режима контроля арифметического переполнения используются директива Q. Соответственно, нашу программу необходимо модифицировать следующим образом, чтобы отловить исключительную ситуацию переполнения:

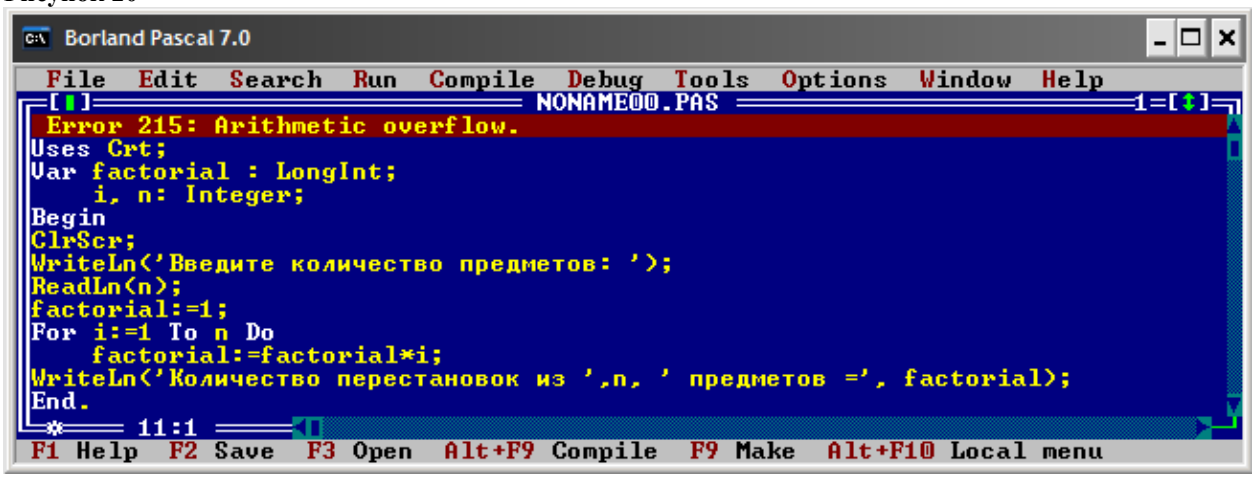
```
{$Q+}
Uses Crt;
Var factorial : LongInt;
  i, n: Integer;
Begin
ClrScr;
WriteLn('Введите количество предметов: ');
ReadLn(n);
factorial:=1;
For i:=1 To n Do
  factorial:=factorial*i;
WriteLn('Количество перестановок из ',n, ' предметов =',
factorial);
End.
```

Теперь, если в результате наших вычислений будут получаться числа, не помещающиеся в отведенный под них тип данных, будет выводиться следующее сообщение об ошибке:

```
Runtime error 215 at 0000:00CC.
```

А если мы запускаем программу из среды Pascal, в верхней части редактора кода будет появляться следующая строчка (см. рисунок)

Рисунок 20



Цикл с предусловием

Не всегда количество итераций (или повторений), которое должен совершить цикл, известно. Например программа должна выполнять какие-либо операции до тех пор, когда пользователь не совершит какие-нибудь действия. Банальные пример: программы-скринсейверы. Они рисуют на экране какие-то замысловатые рисунке до тех пор, пока пользователь не двинет мышкой или не нажмет клавишу на клавиатуре. Поэтому, почти во всех языках программирования были введены специальные циклы, которые повторяют участок кода, до тех пор, пока верно какое-либо условие (например, пока пользователь не нажмет какую-то клавишу). Условие представляют те же самые логические выражения, что мы могли записывать в условном операторе. В них также можно использовать логические операторы **AND, OR, NOT** и т.д. Синтаксис такого цикла в Pascal выглядит следующим образом:

```
While (УСЛОВИЕ) Do
Begin
   {тело цикла}
End;
```

Как реализуется логика выполнения такого цикла? Когда доходит очередь выполняться участку кода, содержащему этот цикл, в первую очередь вычисляется значение логического выражения УСЛОВИЕ и, если оно истинно, выполняются операторы, представляющие собой тело цикла, после чего проверка условия повторяется вновь. И так до тех пор, пока УСЛОВИЕ не станет ложным.

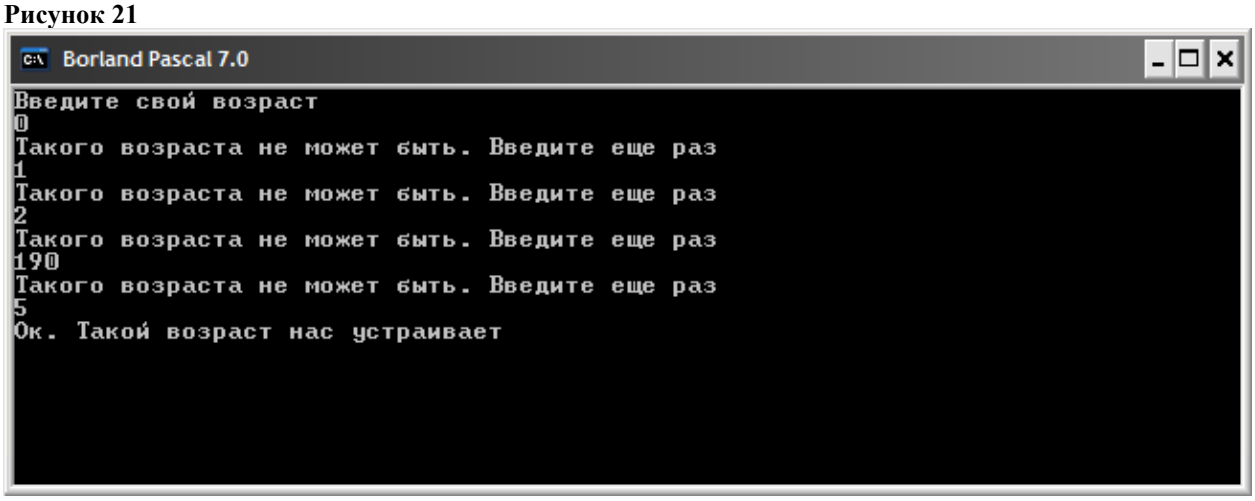
Необходимо использовать такие циклы с предельной осторожностью: гарантия того, что УСЛОВИЕ в некоторый момент все-таки примет ложное значение, целиком лежит на плечах программиста. В противном случае – может получиться бесконечный цикл и программа «зависнет», т.е. перестанет отвечать на любые действия пользователя. Все мы знаем, как неприятно иметь дело с такими ситуациями. Данные, которые пользователь создал или получил при помощи такой программы, могут быть в результате «зависания» программы безвозвратно утрачены.

Приведем пример использования цикла While для гарантии корректного ввода пользователем своего возраста. Предполагаем, что возраст пользователя не может быть

меньше трех лет (вряд ли ребенок до трех лет сможет пользоваться нашей программой) и не может превышать 150 лет[11]. Поэтому, если пользователь введет свой возраст, не попадающий в этот диапазон, мы попросим его ввести заново и так до тех пор, пока не будет получен удовлетворяющий нас результат.

```
Uses Crt;
Var age : Byte;
Begin
ClrScr;
WriteLn('Введите свой возраст');
ReadLn(age);
While (age<3) Or (age>150) Do
Begin
  WriteLn('Такого возраста не может быть. Введите еще раз');
  ReadLn(age);
End;
WriteLn('Ок. Такой возраст нас устраивает');
End.
```

Рисунок 21



Практикум: Программа-screensaver

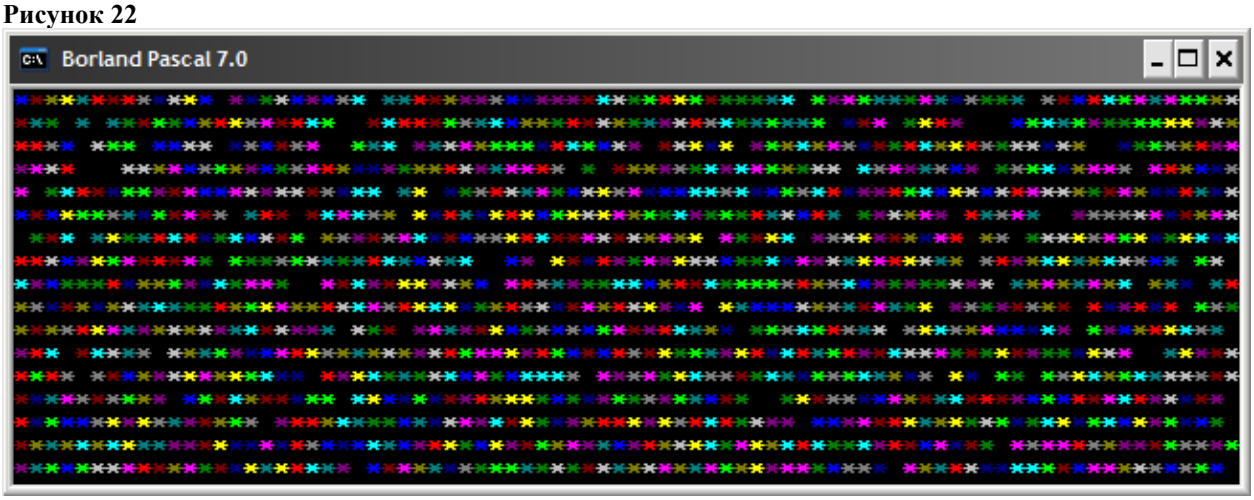
Используя цикл с условием напишем программу, которая чем-то похожа на программы-заставки (скринсейверы – screensavers). Суть ее будет заключаться в том, что до тех пор пока пользователь не нажмет любую клавишу на клавиатуре, мы будем в случайном месте выводить символ звездочка (*) случайного цвета.
Для отлавливания события «пользователь нажал на любую клавишу» воспользуемся процедурой `KeyPressed` из модуля `Crt`, которая возвращает `TRUE`, если пользователь нажал клавишу и `FALSE` в противном случае. Для позиционирования звездочки и изменения ее цвета также будем использовать функционал модуля `Crt` , а для обеспечения случайности цвета и местоположения будем использовать генератор псевдослучайных чисел (см. Дополнительный материал к модулю 1 и модулю 2).

```
Uses Crt;
Begin
ClrScr;
```

[11] При этом официальный зарегистрированный рекорд долголетия принадлежит француженке Жанне-Луизе Кальман, скончавшейся в 1997 году в возрасте 122 лет.

```
Randomize; {Инициализируем генератор случайных чисел}
While Not KeyPressed Do {… пока пользователь не нажал клавишу}
Begin
  GotoXY(Random(79), Random(23)); {в случайное место экрана}
  TextColor(Random(15)); {Случайный цвет}
  Write('*');
End;
End.
```

Рисунок 22



Цикл с постусловием

При использовании цикла While условие проверяется и перед началом итераций, поэтому, если условие ложно, то цикл может ни разу и не выполнится. Кроме того, использование такого вида цикла приводит к повторяемости некоторых участков кода, обеспечивающие истинность условия на первой итерации. Обратимся снова к примеру ввода корректного возраста пользователем. Ведь для того, чтобы условие

```
(age<3) Or (age>150)
```

можно было проверить, необходимо, чтобы возраст был уже введен хотя бы один раз. В теле цикла

```
  WriteLn('Такого возраста не может быть. Введите еще раз');
  ReadLn(age);
```

мы повторяем строчку ввода возраста еще раз. Это простой пример, однако, комплекс мероприятий, обеспечивающие истинность первоначальной проверки условия в цикле может быть значительно сложнее, поэтому такое дублирование крайне не желательно. Для предотвращения такой ситуации используют *циклы с постусловием.* Такие циклы гарантированно выполняют хотя бы одну итерацию, а лишь только после этого вычисляют условие, влияющее на принятие решение о дальнейших повторениях. Синтаксис таких циклов в Pascal следующий:

```
Repeat
  {тело цикла}
Until Условие;
```

Используя такой вид циклов, текст программы, осуществляющей ввод корректного возраста преобразуется в следующий вид:

```
Uses Crt;
Var age : Byte;
Begin
ClrScr;
Repeat
  WriteLn('Введите свой возраст');
  ReadLn(age);
Until (age>=3) and (age<=150)
WriteLn('Ок. Такой возраст нас устраивает');
End.
```

Обратите также внимание, что условие в цикле **Repeat** отличается от условия в цикле **While**. Его значение изменилось на противоположное. Дело в том, что если перевести на русский язык конструкцию цикла с постусловием, то получится следующая фраза: «*Повторять операторы, содержащиеся в теле цикла до тех пор, пока не выполняется УСЛОВИЕ*».

Оператор досрочного прерывания цикла (break)

В некоторых случаях возникает необходимость прервать выполнение цикла, не дожидаясь выполнения *условия останова*, т.е. не дожидаясь выполнения условия окончания итераций. Для этого предусмотрен оператор `break`, который прекращает выполнение цикла и программа после этого переходит к выполнению того участка, который находится после цикла.

Практикум: управление символом на экране при помощи клавиш управления положения курсором

Зададимся целью написать программу, которая помещает некоторый символ в центре экрана, а пользователь, при помощи клавиш «вверх», «вниз», «влево» и «вправо» может управлять местоположением этого символа. Программа должна завершаться по нажатию клавиши ESC, а в случае, если будут достигнуты границы экрана, пользователю будет выведено сообщение об успешном завершении миссии.
Для того, чтобы отловить нажатие функциональных клавиш (ESC, клавиши управления курсором), будем использовать функцию `ReadKey` из модуля `Crt`. Эта функция возвращает код нажатой пользователем клавиши. Нужные нам коды мы можем посмотреть в специальных таблицах.

Клавиша	Код
ESC	27
Вверх	72
Вниз	80
Влево	75
Вправо	77

Для того, чтобы проверить код нажатой клавиши нам потребуется ввести переменную *символьного типа*, т.к. `ReadKey` возвращает символ, соответствующий нажатой клавише. Сама проверка осуществляется следующим образом: символьная переменная сравнивается с кодом, записанным после символа «шарп» (или «решетка») – «#».

```
Uses Crt;
Var C : Char; {Char – символьный тип данных}
Begin
C:=ReadKey;
```

```
If C=#0 Then C:=ReadKey; {Если ReadKey возвратил символ с}
                         {кодом 0, считываем еще раз}
If C=#75 Then WriteLn('Нажата клавиша ВЛЕВО');
End.
```

Проверка

```
If  C=#0 Then C:=ReadKey;
```

необходима для тех случаев, когда `ReadKey` возвращает перед кодом нажатой клавиши нулевой символ.

Движение символа будем осуществлять вычитая либо прибавляя единицу к соответствующей координате. Поскольку вертикальная ось направлена вниз (ось Y), а горизонтальная – слева направо (Ось X), то для движения влево нам необходимо будет уменьшать горизонтальную координату, а для движения вверх – уменьшать координату Y.

Чтобы не оставлять трек движения символа по экрану, будем на каждой итерации очищать экран при помощи процедуры `ClrScr`. (На самом деле, можно очистку экрана и убрать, тоже интересно будет – можно рисовать звездочкой различные рисунки на экране).

Исходный текст программы будет тогда выглядеть следующим образом:

```
Uses Crt;
Var c:char;
    x,y: Byte;
Begin
  x:=40; {Координаты середины экрана}
  y:=12;
  Repeat
     ClrScr;
     GotoXY(x, y);
     Write('*');
     c:=ReadKey;
     If c=#0 Then c:=ReadKey;
     Case c Of
        #72: y:=y-1; {двигаемся вверх}
        #80: y:=y+1; {двигаемся вниз}
        #75: x:=x-1; {двигаемся влево}
        #77: x:=x+1; {двигаемся вправо}
     End;
     If (x=0) Or (x=80) Or (y=0) Or (y=25) Then
     Begin
        Writeln('Миссия завершена!');
        Break;
     End;
Until c=#27; {Выполняем цикл до тех пор пока пользователь}
             {не нажмет клавишу Esc}
End.
```

Оператор безусловного перехода к следующей итерации (continue)

В случае, когда при некоторых условиях необходимо не прекращать выполнение цикла, а начать следующую итерацию, применяется оператор `continue`.

В качестве примера приведем программу, которая суммирует только нечетные числа из тех, что вводит пользователь. В случае, если пользователь введет четное число, благодаря оператору continue, цикл перейдет к следующей итерации, требуя от пользователя ввода

следующего значения. Программа выполняется, пока общее количество вводимых чисел не превысит 10.

Четность или нечетность числа проверяется по остатку деления его на 2. Если остаток от деления равен нулю, значит число четное, в противном случае – нечетное. Для вычисления остатка от деления воспользуемся оператором **Mod**.

```
Var Number, i, Sum, cnt: Integer;
Begin
Sum:=0;
For i:=1 To 10 Do
Begin
     ReadLn(Number);
     If Number Mod 2=0 Then Continue;
     Sum:=Sum+Number;
     cnt:=cnt+1;
End;
WriteLn('Сумма нечетных чисел= ', sum, ' Их количество=',cnt);
End.
```

Модуль 4. Одномерные массивы

В прошлой главе мы научились суммировать большое количество значений, вводимых пользователем. Нам не нужны были сами значение как таковые. Поэтому мы объявили одну переменную и прибавляли к ней значение по мере ввода пользователем новых данных. Но как быть, если все вводимые пользователем значения необходимо запоминать для дальней обработки? Ведь значения могут водиться не пользователем, а считываться, например, из файла, вычисляться, передаваться с различных устройств по USB, Ethernet-порту или другими способами.
Для хранения таких объемов данных предусмотрены особые структуры в языках программирования – *массивы*.

Массив – это упорядоченное множество однотипных элементов.

Упорядоченность означает, что все элементы в массиве имеют свой номер (он называется *индексом*), по которому осуществляется доступ к ним (определение значения элемента, или его запись). А однотипность означает, что все элементы имеют один и тот же тип данных.
В Pascal общая схема объявление массивов выглядит следующим образом:

```
Var ИМЯ_МАССИВА: Array [НЗ..КЗ] Of ТИП_ДАННЫХ;
```

Здесь,

ИМЯ_МАССИВА – это идентификатор, по которому в дальнейшем можно обращаться к массиву. Подчиняется всем правилами именования переменных (см. Модуль 1 – требования к именам идентификаторов).

НЗ – начальное значение при нумерации элементов массива (начальное значение *индекса* массива). Может быть только целым числом. Начальное значение не может быть больше конечного значения.

КЗ – конечное значение при нумерации элементов массива. Конечное значение не может быть меньше начального значения.

ТИП_ДАННЫХ – тип данных, который будут иметь все элементы массива.

Для облегчения понимания сути массивов, полезно представлять себе его как таблицу, состоящей из одной строчки, каждая ячейка которой имеет свой номер (индекс), а значения, записанные в ячейке, символизируют значения элементов массива.
Например,

Значение	**34**	**12**	**244**	**64**	**-8**
Индекс	**1**	**2**	**3**	**4**	**5**

Здесь значение ячейки под номером 2 будет 12, значение с индексом 4 = 64 и т.д.
Приведем примеры объявления массивов

Пример 1. Массив из 5 целочисленных элементов

```
Var intArray : Array[1..5] Of Integer;
```

Здесь номер первого элемента равен 1, а номер последнего элемента – 5.

Пример 2. Массив из 6 вещественных элементов

```
Var fltArray: Array[0..5] Of Real;
```

В приведенном примере нумерация элементов массива начинается с 0, поэтому номер последнего (шестого) элемента будет 5.

Обратите также внимание, что для указания диапазона изменения индекса используется знак двоеточия «..», в котором не предусматривается никаких разделяющих символов (пробелов, табуляции, переноса строки).
После того, как массив объявлен в разделе деклараций, с его элементами можно осуществлять операции точно таким же образом, что и с обычными переменными такого же типа. Например, если массив объявлен следующим образом:

```
Var massiv: Array [1..3] Of Integer;
```

то его элементы пользователи могут вводить с клавиатуры, например, так:

```
ReadLn(massiv[1]);
ReadLn(massiv[2]);
```

Т.е. указание индекса элемента массива записывается после его имени в квадратных скобочках.
С элементами массива можно выполнять все операции, которые присущи переменным того же типа, что его элементы:

```
massiv[3]:=0;
massiv[3]:=massiv[1]+massiv[2];
If (massiv[2]>10) Then
  WriteLn('Элемент 2 равен', massiv[2]);
```

О выходе значения индекса за допустимый диапазон

Рассмотрим ситуацию, когда максимальный индекс массива равен 10, т.е. некоторый массив объявлен следующим образом:

```
Var a: Array [1..10] Of Integer;
```

а мы попытаемся работать с элементами, имеющими индекс, превышающий 10, например следующим образом:

```
A[11]:=0;
```

Что в таком случае произойдет? А произойдет следующее. Компилятор определит границы допустимого диапазона для индекса этого массива (в нашем случае – это числа от 1 до 10) и проверит попадание указанного индекса (у нас – это 11) в допустимый диапазон. Поскольку число 11 не попадает в диапазон от 1 до 10, то компиляция программы завершится с ошибкой

```
Error 76: Constant out of range.
```

что означает «Константа вышла за допустимый диапазон».
Но ведь индекс массива не всегда известен на этапе компиляции. Порой индексы массива вычисляются по определенным формулам или могут вводиться пользователем. Например, приведенная ниже программа без каких-либо ошибок будет скомпилирована:

```
Var a : Array[1..10] Of Integer;
    I : Integer;
```

```
Begin
  ReadLn(i);
  a[i]:=10;
End.
```

Если пользователь введет в переменную i значение, не попадающее в интервал от 1 до 10, то программа, очевидно, будет работать некорректно. Однако, как и в случае ошибок переполнения, никаких сообщений об ошибках выведено не будет. Следовательно, такая ситуация может привести к непредсказуемости поведения больших программ, и программисту следует обращать особое внимание на то, чтобы все индексы гарантированно попадали в допустимый диапазон. Для того, чтобы облегчить задачу программиста, мы можем при помощи директивы компилятора включить контроль такого рода исключительных ситуаций. Директива эта – {$R+}. Добавим эту директиву в предыдущий код:

```
{$R+}
Var a : Array[1..10] Of Integer;
    I : Integer;
Begin
  ReadLn(i);
  a[i]:=10;
End.
```

Теперь, после того, как пользователь введет некорректный индекс, программа завершит свою работу с ошибкой времени выполнения

```
Runtime error 201: Range check error.
```

На этапе разработки и отладки программ, использующих массивы, рекомендуется всегда включать контроль попадания индекса в допустимый диапазон.

Типовые задачи, связанные с массивами

При работе с массивами, часто возникает необходимость в выполнении следующих операций над ними:

- ввод и вывод элементов массива;
- подсчет суммы элементов массива;
- нахождение минимального и максимального элемента в массиве;
- сортировка элементов массива.

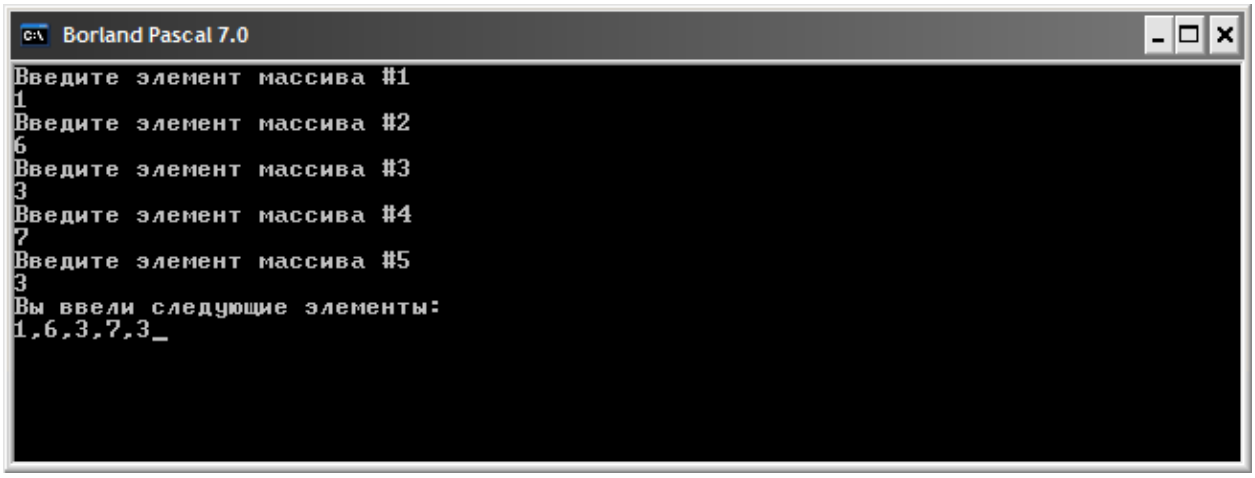
Рассмотрим каждую из этих операций подробно

Ввод и вывод элементов массива

Все операции по вводу и выводу элементов массива осуществляются поэлементно. Для этого часто используются циклы, перебирающие весь допустимый диапазон индексов массива.
Например:

```
Uses Crt;
Var inputArray: Array[1..5] Of Integer;
    i: Integer;
Begin
  ClrScr;
  {Ввод элементов массива}
```

```
  For i:=1 To 5 Do
  Begin
    WriteLn('Введите элемент массива #', i);
    ReadLn(inputArray[i]);
  End;
  {Вывод элементов массива}
  WriteLn('Вы ввели следующие элементы: ');
  For i:=1 To 5 Do
    If i<>1 Then
       Write(',', inputArray[i])
    Else
       Write(inputArray[i]);
End.
```



В данном примере элементы массива выводятся в строчку, разделяемые запятыми. Поскольку перед первым элементом массива ставить запятую не требуется, то в цикл был добавлен условный оператор, проверяющий номер элемента массива и, в зависимости от того, равен он единице или нет, выводящий очередной элемент массива с запятой или без нее.

Подсчет суммы всех элементов массива

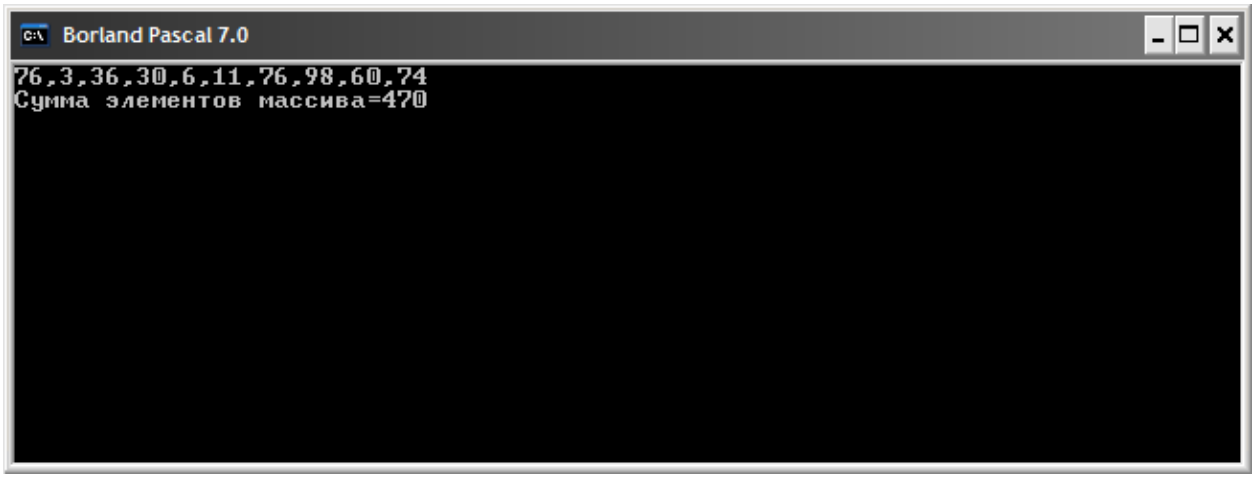
Зачастую в массиве хранятся элементы, которые необходимо обрабатывать статистически: находить сумму всех элементов, находить максимальный или минимальный элементы, вычислять среднее значение.

В этом параграфе мы рассмотрим метод подсчета суммы элементов массива, которые генерируются автоматически при помощи генератора псевдослучайных чисел, на характер подсчета суммы это никак не отразится. Аналогичным образом суммировать элементы массива можно и в случае ручного ввода значений.

```
Uses Crt;
Var randomArray: Array[1..10] Of Integer;
    i, Sum: Integer;
Begin
  ClrScr;
  Randomize;
  For i:=1 To 10 Do
  randomArray[i]:=random(100); {Генерируем элементы массива}
                               { в диапазоне от 0 до 100}
  {Выведем элементы массива на экран}
```

```
  For i:=1 To 10 Do
      If i<>1 Then
          Write(',', randomArray[i])
      Else
          Write(randomArray[i]);
  {Вычисляем сумму элементов}
  Sum:=0;
  WriteLn;
  For i:=1 To 10 Do
       Sum:=Sum+randomArray[i];
  WriteLn('Сумма элементов массива=', Sum);
End.
```

В данном примере, сумма элементов массива накапливается в переменной `Sum`, которую перед началом суммирования обнуляют, чтобы не исказить сумму случайно записанным в нее значением.



Нахождение минимального и максимального элемента в массиве

Алгоритм поиска минимального и максимального значений в массиве является пошаговым. На первом шаге в качестве искомого значения (максимального или минимального) принимается первый элемент массива. На последующих шагах это значением сравнивается с другими элементами массива и в случае, если очередной сравниваемый элемент является больше (меньше), его значение снова принимается за искомое. Эта операция повторяется до тех пор, пока не будут перебраны все элементы массива.

Рассмотрим работу этого алгоритма на примере следующего массива из 5-ти элементов:

```
7, 6, 13, 45, 4
```

Будем искать максимальное значение в этом массиве. Для этого, будем использовать переменную `Max`, которую на первом шаге приравняем первому элементу в массиве:

Номер шага	Значение переменной Max	Проверяемое условие
1	Max=7	
2	Max=7	Проверяем неравенство 6>Max: условие не выполняется, поэтому значение переменной Max остается без изменения
3	Max=7	Проверяем условие 13>Max: условие выполняется, поэтому приравниваем

		переменную Max:=13
4	Max=13	Проверяем условие 45>Max: условие верно, поэтому изменяем значение переменной Max:=45
5	Max=45	Проверяем условие 4>Max: условие не выполняется, переменная Max остается без изменений.

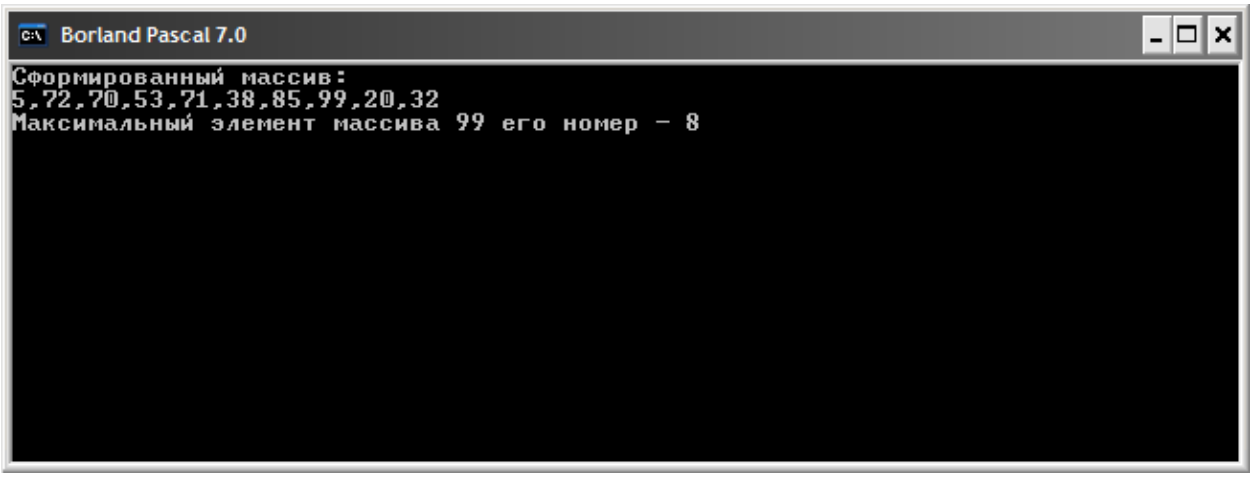
Таким образом, после последнего шага алгоритма, переменная Max хранит максимальное значение элементов массива.

Для поиска минимального элемента, шаги будут теми же самыми, изменится только знак проверяемого неравенства – он станет противоположным.

Текст программы, реализующий данный алгоритм следующий (массив заполняется псевдослучайными значениями). Помимо поиска максимального элемента, будем также запоминать его положение в массиве (его номер):

```
Uses Crt;
Var randomArray: Array[1..10] Of Integer;
  Index, Max, i: Integer;
Begin
ClrScr;
Randomize;
For i:=1 To 10 Do
  randomArray[i]:=random(100);
{Выведем массив на экран, для контроля}
WriteLn('Сформированный массив: ');
For i:=1 To 10 Do
  If i<>1 Then
       Write(',', randomArray[i])
  Else
       Write(randomArray[i]);
WriteLn;
{Ищем максимальный элемент и его номер}
Index:=1;
Max:=randomArray[1];
For i:=2 To 10 Do
  If randomArray[i]>Max Then
       Begin
            Max:=randomArray[i];
            Index:=i;
       End;
WriteLn('Максимальный элемент массива ', Max, ' его номер – ',
Index);
End.
```

Результат работы программы приведен на рисунке ниже.



Сортировка элементов массива

Сортировка массива – это изменение порядка его элементов таким образом, чтобы они располагались либо строго по возрастанию, либо строго по убыванию. Необходимость этой операции объясняется прямыми задачами (например, расстановка списка фамилий в алфавитном порядке), так и нуждами в быстром поиске среди большого объема данных. Если элементы отсортированы, то поиск осуществлять намного проще, чем в хаотически расположенных элементах. Так, если мы имеем дело с отсортированным по возрастанию массивом, то его минимальный элемент будет первым, а максимальный – последним. Существует большое разнообразие алгоритмов сортировки. Каждый из них отличается типом данных, которые необходимо сортировать, и временем (в зависимости от количества элементов), которое им в худшем случае приходится затратить на полную сортировку.

В настоящем параграфе мы рассмотрим самый простой в реализации метод, который получил название «сортировка вставками». Суть его заключается в сравнении каждого элемента с каждым и, если проверяемое неравенство выполняется, сравниваемые элементы меняются местами. Вычислительная сложность этого алгоритма довольно большая. В общем случае ему потребуется сделать $\sim n^2$ операций. Если n (это количество элементов массива) достаточно велико, то алгоритм работает крайне медленно.

В качестве примера рассмотрим сортировку массива, состоящего из трех элементов

```
2, 5, 1
```

Сортировать будем по возрастанию.

На первом шаге зафиксируем первый элемент и будем сравниваем его (число 2) со вторым (число 5), поскольку второе число больше, то никаких действий не требуется – между ними установлено правильное соотношение. На следующем шаге – сравниваем зафиксированный элемент со следующим числом – числом 1. Поскольку 1<2, то меняем эти элементы местами, получив следующий массив:

```
1, 5, 2
```

Таким образом, после первого шага, мы переместили наименьший элемент в начало. Теперь, мы можем повторить операцию для оставшейся части массива, т.е. выполнив те же самые последовательные сравнения второго элемента со всеми оставшимися. В итоге – получим полностью отсортированный массив.

Исходный текст, реализующий такой метод сортировки, приведем ниже:

```
Uses Crt;
Var randomArray: Array[1..10] Of Integer;
    temp, i,j: Integer;
Begin
Randomize;
```

```
ClrScr;
WriteLn('Исходный массив:');
{Генерация и вывод исходного массива}
For i:=1 To 10 Do
Begin
    randomArray[i]:=random(100);
    If i<>1 Then
       Write(',',randomArray[i])
    Else
       Write(randomArray[i]);
End;
{Сортировка массива}
For i:=1 To 10 Do
    For j:=i To 10 Do
        If randomArray[j]<randomArray[i] Then
           Begin
              temp:=randomArray[i];
              randomArray[i]:=randomArray[j];
              randomArray[j]:=temp;
           End;
WriteLn;
WriteLn('Отсортированный массив:');
{Вывод отсортированного массива}
For i:=1 To 10 Do
    If i<>1 Then
       Write(',',randomArray[i])
    Else
        Write(randomArray[i]);
End.
```

Обратите внимание, что мы впервые использовали вложенные циклы: т.е. один цикл «крутится» внутри другого. Первый цикл (по переменной `i`) фиксирует элемент массива, который надо сравнить с другими элементами, а вложенный цикл (по `j`) перебирает оставшиеся элементы, чтобы их можно было сравнить с `i`-ым элементом.

В этом примере, мы использовали переменную `temp` для того, чтобы поменять местами элементы массива, для которых выполняется неравенство `randomArray[j]<randomArray[i]`:

- сначала «спасли» значение `i`-го элемента массива, т.е. записали его значение в переменную `temp`;
- вместо значения `i`-го элемента массива записываем значение `j`-го элемента;
- вместо `j`-го элемента записываем значение, которое раньше было в `i`-ом элементе, т.е. значение переменной `temp`.

Для сортировки массива в обратном порядке, достаточно поменять знак неравенства `randomArray[j]<randomArray[i]` на противоположный.

Об особенностях объявления массивов

В Pascal существует особенность при объявлении массивов. Если объявлены два массива

```
Var A: Array[1..10] Of Byte;
    B: Array[1..10] Of Byte;
```

то, с точки зрения синтаксиса, переменные `A` и `B` имеют различные типы, несмотря на то, что они объявлены одинаково. Так, если возникнет необходимость приравнять один массив другому, т.е. выполнить следующую операцию присваивания:

```
A:=B;
```

то синтаксический анализатор сообщит об ошибке: `Error 26: Type mismatch` (Несовместимость типов). Для того, чтобы избежать такой ситуации, одинаковые массивы следует объявлять в одном блоке раздела декларации:

```
Var A, B: Array[1..10] Of Byte;
```

На практике же, обычно создают свой собственный тип данных и объявляют переменные как переменные этого пользовательского типа данных. Такой подход значительно более гибкий, поскольку объявление может встречаться в различных частях программы (это особенно актуально при использовании в программе собственных функций и процедур). Поэтому, мы рекомендуем объявлять массивы так, как это будет описано в настоящем параграфе. Для этого нам потребуется рассмотреть некоторые дополнительные элементы языка Pascal: пользовательские типы и константы.

Пользовательские типы данных

Пользовательские типы данных объявляются в разделе деклараций и начинаются с ключевого слова **Type**:

```
Type ИМЯ_ТИПА=ОПИСАНИЕ_ТИПА;
```

например, мы можем определить следующий свой тип данных:

```
Type TMyType=Integer;
```

После объявление собственного типа данных следует объявить переменную этого нового типа:

```
Var MyVar: TMyType;
```

Удобство заключается в том, что при помощи такого подхода можно одним махом изменить тип данных нескольких переменных. Посмотрите на код, приведенный ниже

```
Type
  TInteger=SmallInt;
Var
  a: TInteger;
  b: TInteger;
  c: TInteger;
```

Достаточно изменить в нем описание типа `TInteger`, например, изменить тип `SmallInt` на `LongInt`, чтобы типы данных переменных `a`, `b`, `c` сразу поменялись.
В случае массивов, объявление одинаковых переменных тогда может выглядеть следующим образом

```
Type
  TArray = Array[1..10] Of Byte;
Var
  A: TArray;
  B: TArray;
```

Теперь, с точки зрения синтаксического анализатора, переменные `A` и `B` имеют одинаковый тип данных.
Тем не менее, даже при таком способе объявления массивов остается еще одно неудобство, связанное с диапазоном индексов, используемым в объявлении массивов. Как правило, если программист использует массивы, то возникает необходимость в организации циклов, пробегающихся по всему диапазону индексов. В случае, если потребуется изменить размерность массива (т.е. изменить количество элементов в нем), программисту придется пересмотреть весь исходный текст и поменять во всех циклах диапазон изменения параметров цикла, чтобы они не выходили за рамки изменения индекса массива. Разумеется, это может быть источником ошибок, так как велика вероятность того, что будут замечены не все циклы, в которых происходит перебор элементов массива.

Чтобы минимизировать количество ошибок, связанных изменениями размерности массива, используют специальные конструкции языка, которые называются *константами*.

Константы

Константы – это особый вид значений, которые не могут изменять своего значения во время работы программы. Значение констант задается на этапе создания программ непосредственно в исходном тексте, и не может быть изменено ни в какой ее части. Объявление констант начинается с раздела деклараций и, как правило, находится перед объявлением переменных. Это обусловлено тем соображением, что некоторые переменные, в частности – массивы, могут использовать значения констант, поэтому они должны быть объявлены раньше таких переменных. Например:

```
Const
  ONE=1;
  NINE=9;
  TEN=ONE+NINE;
Var
  MyArray: Array[ONE..TEN] Of Integer;
```

Считается хорошим тоном давать имена константам в верхнем регистре (т.е. большими буквами), чтобы их можно было легко отличить от переменных в исходном коде.

Рекомендуемый способ объявления массивов

Теперь, объединяя все полученные знания о конструкциях языка Pascal, приведем рекомендуемую схему объявления любых массивов:

1. Объявить константы, определяющие диапазон индексов.
2. Определить собственный тип данных для массива, использующий для определения диапазона объявленные на предыдущем шаге константы.
3. Объявить переменные созданного типа данных.

Например:

```
Const
  LOW_BOUND=0;
  UP_BOUND=10;
Type
  MyArray=Array[LOW_BOUND..UP_BOUND] Of Integer;
Var
  A, B : MyArray;
```

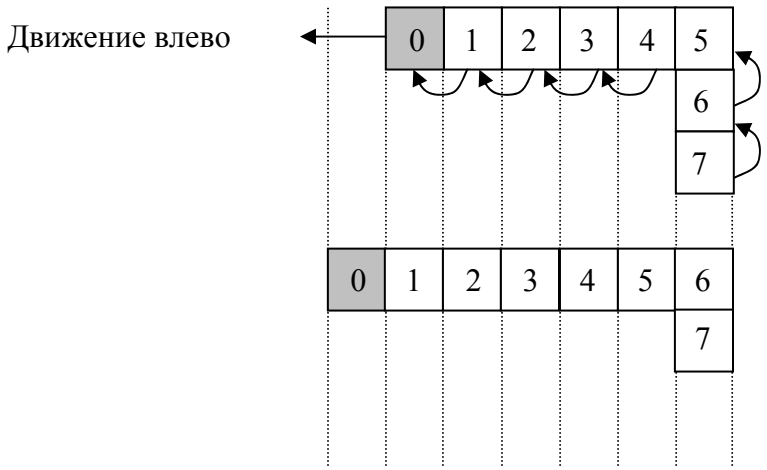
При таком способе объявления массивов, можно будет организовывать циклы по всем их элементам, используя эти константы

```
For i:=LOW_BOUND To UP_BOUND DO
…
```

В связи с этим, разработка и отладка программы значительно облегчится, так как если в результате модификации программы потребуется изменить диапазон изменения индексов массива, то будет достаточно изменить значения констант `LOW_BOUND` и `UP_BOUND`. Вносить изменения в код при этом не потребуется.
В дальнейшем, в наших примерах, мы будем использовать только такой способ объявления массивов.

Практикум: Разработка приложения «Змейка»

В этом параграфе мы используем массивы для разработки приложения, в котором пользователь может управлять местоположением «змейки» на экране.
Тело змейки будет состоять из каких-либо символов, перемещаемых вслед за ведущим символом – головой змеи.
При перемещении, нам, по сути дела, потребуется изменять координату только головы змеи – все остальные части тела будут лишь «перебираться» на предыдущие положения впередистоящих звеньев.



Таким образом, алгоритм движения змейки будет заключаться в следующем:

1. Вычислить новые координаты головы змеи (на нашем рисунке – это нулевой элемент) по следующему правилу:
   - Если была нажата кнопка «ВВЕРХ», то вычитаем из вертикальной координаты Y единицу. Координата X остается прежней.
   - Если была нажата кнопка «ВНИЗ», то прибавляем к вертикальной координате Y единицу. Координата X также остается без изменений.
   - Если была нажата кнопка «ВЛЕВО», то вычитаем единицу из координаты X.
   - Если была нажата кнопка «ВПРАВО» то прибавляем единицу к координате X.
2. Переместить первое звено на место предыдущего нулевого звена, второе звено – на место предыдущего первого и т.д.

Отлавливать нажатие клавиш управления курсором мы будем при помощи процедуры `ReadKey` из модуля `Crt`.
Координаты X и Y будем хранить в двух различных массивах. Длина змейки будет определяться количеством элементов в этих массивах.

```
Uses Crt;
Const
  SNAKE_LEN=6;
Type
  TSnakeCoords=Array[0..SNAKE_LEN] Of Byte;
Var
  X, Y : TSnakeCoords;
  c : Char;
  i : Integer;
  NewX, NewY : Integer;
Begin
{первоначальное положение змейки - левый верхний угол}
For i:=0 To SNAKE_LEN Do
Begin
```

```
  X[i]:=i+1;
  Y[i]:=1;
End;
{Основной цикл – до тех пор пока не нажмут Esc}
Repeat
  ClrScr;
  {Вывод на экран змейки}
  For i:=0 To SNAKE_LEN Do
  Begin
       GotoXY(X[i], Y[i]);
       Write('*');
  End;
  c:=ReadKey;
  If c=#0 Then c:=ReadKey;
  {Получаем местоположение головы змейки}
  NewX:=X[0];
  NewY:=Y[0];
  {На их основе вычисляем новые координаты}
  Case C Of
       #72: NewY:=NewY-1;
       #80: NewY:=NewY+1;
       #75: NewX:=NewX-1;
       #77: NewX:=NewX+1;
  End;
  {Перебрасываем звенья на положения впередистоящих участков}
  For i:=SNAKE_LEN DownTo 1 Do
       Begin
            X[i]:=X[i-1];
            Y[i]:=Y[i-1];
       End;
  {Записываем новые координаты головы змеи}
  X[0]:=NewX;
  Y[0]:=NewY;
Until c=#27;
End.
```

Модуль 5. Строки и многомерные массивы

Представление строк в Pascal

Мы уже сталкивались с типом данных, способным хранить один символ. Это тип данных `Char`. Но в прикладных задачах часто приходится иметь дело не с одним символом, а с целыми словами и предложениями (вспомним хотя бы текстовый редактор). Можно, конечно, работать со строками, объявив массив символов, например, так:

```
Const
  String_Len=100;
Type
  StringType=Array[1..StringLen] Of Char;
Var
  MyString: StringType;
```

Но такой подход зачастую неудобен, поскольку требует непосредственного написания всех процедур манипулирования со строками (копирование строчек, поиск подстроки, извлечение подстроки и т.д.). Поэтому был введен специальный тип данных для работы со строками: `String`. По сути дела, строковый тип данных – это тот же массив символов, однако для него предусмотрены специальные процедуры и функции, облегчающие разработку программ.

Максимальная длина строки для типа данных String ограничена 255-ю символами. Это ограничение было снято в реализации языка Object Pascal, используемой в среде Delphi. Все символы строки нумеруются от 1 до длины строки. В нулевом символе хранится символ, с кодом, равным длине строки. Так, например, если строка содержит 32 символа, то нулевой символ будет пробелом (' '), так как его код равен 32.

В остальном, использование переменных строкового типа ничем не отличается от использования тех переменных, которые мы использовали ранее. Приведем пример:

```
Var
  UserName : String;
Begin
  WriteLn('Введите ваше имя');
  ReadLn(UserName);
  WriteLn('Рады приветствовать, ', UserName);
End.
```

Эта программа просит ввести пользователя его имя, и затем просто выводит приветственное сообщение, содержащее обращение по имени к пользователю.

Операции над строками

Для строковых переменных определены операции присваивания (для нее используется уже знакомый нам оператор ':='), сравнения ('=' – равенство, '<' – меньше, '>' – больше, '<=' – меньше или равно, '>=' – больше или равно) и *конкатенации* – объединения строк. Для конкатенации используется символ «+», который, несмотря на то, что используется для сложения, не подразумевает независимость результат от мест «слагаемых»-строчек. Действительно, если у нас есть две строки

```
S1:='Маша';
S2:=' ела кашу.';
```

то в результате конкатенации S1+S2 мы получим строчку «Маша ела кашу», а поменяв «слагаемые» местами – « ела кашу.Маша».

Сравнение строк происходит следующим образом: если одна из сравниваемых строк имеет большую длину, то она считается больше той, что меньше по длине. Если длины строк одинаковы, то происходит посимвольное сравнение – знак неравенства будет

совпадать со знаком неравенства несовпадающих символов. При этом символ в нижнем регистре считается больше соответствующего символа в верхнем регистре (например, 'm'>'M'). Общее же правило при сравнении символов такое: среди двух сравниваемых символов тот из них считается больше, чей код является больше. Приведем примеры сравнения строк

Строка 1	Строка 2	Результат сравнения
'Pascal'	'pascal'	'pascal'>'Pascal'
'Turbo Pascal'	'Pascal'	'Turbo Pascal'>'Pascal'
'abcdef'	'acdeff'	'abcdef'<'acdeff'

Для того, чтобы узнать длину строки, хранящейся в переменной строкового типа, можно использовать два подхода:

1. Вычислить код символа, находящегося на нулевой позиции, например

```
Var S: String;
    L: Byte;
Begin
     WriteLn('Введите строку');
     ReadLn(S);
     L:=Ord(S[0]);
     WriteLn('Вы ввели строку, длиной ', L);
End.
```

   В этом примере мы использовали функцию `Ord` для получения кода нулевого символа строки `S[0]`.

2. Использовать функцию `Length`.

```
Var S: String;
     L: Byte;
Begin
     WriteLn('Введите строку');
     ReadLn(S);
     L:=Length(S);
     WriteLn('Вы ввели строку, длиной ', L);
End.
```

Аналогично первому способу определения длины строки, можно узнать какой символ находится на определенной позиции – для этого достаточно рассматривать строковую переменную как массив символов. Как правило, этим пользуются, когда необходимо «пробежаться» по всем символам строки для реализации различных алгоритмов над текстами (например, шифрования, когда нужно каждый символ или последовательность заменить другим символом).

Попробуем применить полученные знания для подсчета количества гласных и согласных символов в строке, состоящей из букв английского алфавита.

Для этой цели нам потребуется в введенной строке проверить каждый символ на принадлежность множеству гласных или согласных символов. Символы строки будем перебирать в цикле при помощи параметра, изменяющегося от 1 до длины строки. Поскольку при проверке строк регистр символов важен, то, чтобы нам отдельно не проверять на равенство верхнему и нижнему регистру символов, приведем проверяемые символы к верхнему регистру при помощи функции `UpCase` (она не работает для

русских символов). Количество гласных символов в строке будем подсчитывать в переменной `glasCnt`, а согласных – в `soglasCnt`.

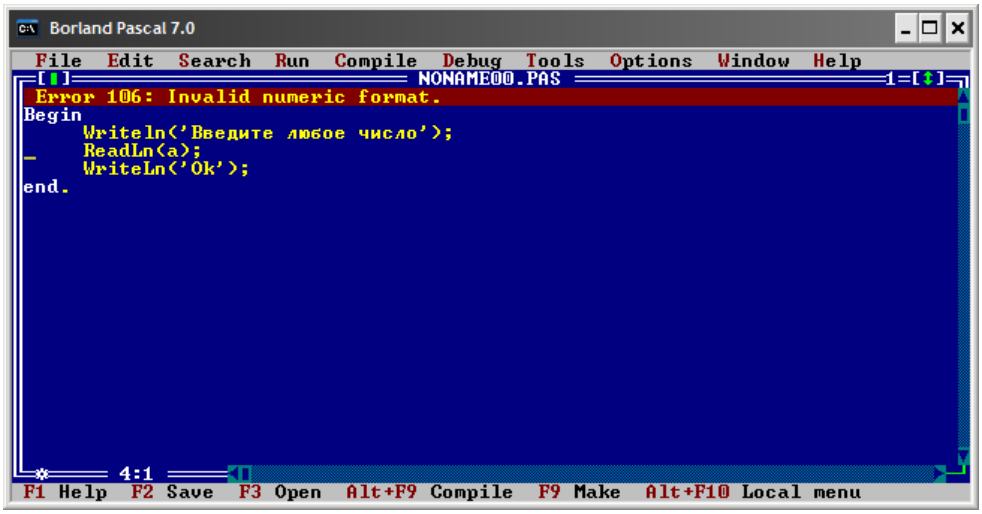
```
Var InputStr: String;
    i, glasCnt, soglasCnt : Integer;
Begin
WriteLn('Введите фразу на английском языке');
ReadLn(InputStr);
glasCnt:=0;
soglasCnt:=0;
For i:=1 To Length(InputStr) Do
  Case UpCase(InputStr[i]) Of
       'A','E','I','O','U','Y': glasCnt:=glasCnt+1;
       Else
            Case UpCase(InputStr[i]) Of
                 'A'..'Z': soglasCnt:=soglasCnt+1;
            End;
  End;
WriteLn('Гласных символов=', glasCnt, ' Согласных=',
soglasCnt);
End.
```

Практикум: Проверка корректности ввода чисел. Преобразование строки в число

Во многих наших программах пользователю необходимо вводить какие-либо числовые данные. Но ведь никто не запретит ему вместо корректного числа ввести любую последовательность чисел. Что произойдет в этом случае с нашей программой? Давайте рассмотрим следующий простой код:

```
Var a: Integer;
Begin
  WriteLn('Введите любое число');
  ReadLn(a);
  WriteLn('OK');
End.
```

Запустим его, а после приглашения ввести число, введем любую строчку. Заметим, что программа «вылетела» с сообщение об ошибке времени выполнения «Error 106: Invalid numeric format» - «Неправильный числовой формат».
Соответственно, все дальнейшие действия программа выполнять не будет, а досрочно «аварийно» завершит свою работу.
Как можно написать программу, устойчивую к ошибкам ввода чисел пользователями? Можно предложить следующий способ: все данные пользователь должен будет вводить в строковые переменные, а мы уже будет проверять, являются ли данные, содержащиеся в строке корректным числом. В случае, если строка представляет собой правильное число, то мы его при помощи процедуры `Val` преобразуем в числовую переменную.
Процедура `Val` используется следующим образом:




```
Var S: String;
    R, Code: Integer;
Begin
…
Val(S, R, Code);
…
```

Т.е. первый аргумент – это исходная строка, содержащая число в виде текста, второй аргумент – это числовая переменная-назначение, куда будет помещено преобразованное в число значение, а третий аргумент (у нас он называется Code) – это признак корректности преобразования. Если строка не содержит никаких «лишних» символов, то значение этой переменной будет равно нулю, в противном случае – будет содержать номер некорректного символа.

Таким образом, для того, чтобы обезопасить программу от ввода некорректных чисел, можно использовать следующий код:

```
Uses crt;
Var
   S: String;
   a, code: Integer;
Begin
     ClrScr;
     Writeln('Введите любое число');
     ReadLn(S);
     Val(S,a, Code);
     If Code<>0 Then
        WriteLn('Ошибка! Номер некорректного символа:',Code)
     Else
         WriteLn('Число введено корректно');
End.
```

Практикум: Подсчет слов в предложении

Цель этого практикума написать программу, считающей количество слов в введенной пользователем строке. Словом будем считать любой символ, окруженный пробелами с двух сторон.

Первое что приходит на ум, что для реализации этой программы достаточно посчитать количество пробелов и прибавить единицу. Это будет верно только в том случае, если

пользователь будет аккуратно разделять слова одним пробелом. Рассчитывать на это было бы наивным, поэтому при разработке программы будем считать, что слова могут разделяться любым количеством пробелов. В таком случае алгоритм подсчета количества слов будет состоять из следующих шагов:

- Будем перебирать последовательно все символы в строке, от первого до последнего, и проверять следующее условие:
  - если очередной проверяемый символ является пробелом, а предыдущий символ не является таковым, то прибавляем к счетчику слов единицу.

У предложенного алгоритма есть две крайние ситуации:

1. Для первого символа не имеет смысла проверить предыдущий символ.
2. Последний символ может и не быть пробелом, но тогда предложенный алгоритм не учтет последнее слово.

Поэтому проверка этих условий должна включаться в общую схему алгоритма (проверки `i<>1` и `i=Length(s)` в приведенном ниже коде).

```
Var
    s : String;
    i,cnt : Integer;
Begin
  WriteLn('Введите предложение');
  ReadLn(s);
  cnt:=0;
  For i:=1 To Length(s) Do
       If (s[i]=' ') And (i<>1) Or (i=Length(s)) Then
            If s[i-1]<>' ' Then cnt:=cnt+1;
  WriteLn('Количество слов :', cnt);
End.
```

Функции для работы со строками

Для удобства работы со строковым переменными в синтаксис языка Pascal были введены некоторые функции, облегчающие работу со строками

Copy

Позволяет извлечь подстроку заданной длины и начиная с заданной позиции из исходной строки. Аргументами этой функции являются:

- исходная строка;
- номер символа, с которого необходимо извлечь подстроку;
- длина извлекаемой подстроки.

Например, в результате выполнения следующего кода

```
Var S:String;
Begin
  S:='Маша ела кашу';
  S:=Copy(S, 1, 4);
  WriteLn(S);
End.
```

будет выведено слово ‘Маша’, поскольку мы вырезали из исходной строки четыре символа, начиная с первого.

Pos

Ищет первое упоминание подстроки в строке и возвращает номер символа, начиная с которого заданная подстрока была найдена. Если подстрока найдена не было, функция возвращает в качестве своего значения 0.
Аргументами этой функции являются:

- подстрока, которую необходимо найти;
- строка, в которой производится поиск.

В следующем примере мы из строки, содержащей адрес некоторого документа в Internet, будем извлекать адрес сервера, на котором этот документ находится. Т.е. если строка будет содержать адрес «http://www.specialist.ru/school/Default.aspx», то наша программа должна вернуть строку «www.specialist.ru». Функцию Pos будем использовать для поиска подстроки «://», после которой начинается адрес сервера и для поиска первого одинарного слеша «/», перед которым этот адрес заканчивается.

```
Const CDELIM='://';
Var Url, Server: String;
  delim:Byte;
Begin
  WriteLn('Введите адрес документа (URL):');
  ReadLn(Url);
  delim:= Pos(CDELIM, Url);
  Url:=Copy(Url, delim+Length(CDELIM), Length(Url)-delim);
  Url:=Copy(Url, 1, Pos('/', Url)-1);
  WriteLn('Сервер:' ,Url);
End.
```

Insert

Позволяет вставить заданную подстроку в строку, начиная с указанной позиции. Если после вставки подстроки, длина получающейся строки превышает 255 символов, то все символы после 255-го будут отброшены.
Аргументами этой процедуры являются:

- подстрока, которую необходимо вставить;
- исходная строка, содержание которой необходимо изменить;
- позиция, перед которой необходимо вставить фрагмент строки.

Следующий код

```
Var S:String;
Begin
  S:='Маша кашу';
  Insert(' ела', S, 5);
  WriteLn(S);
End.
```

выведет на экран строку ‘Маша ела кашу’.

Delete

Удаляет заданное количество символов из исходной строки, начиная с указанной позиции.
Аргументами этой процедуры являются:

- исходная строка, из которой необходимо удалить символы;
- позиция, начиная с которой необходимо удалить символы;
- количество удаляемых символов.

В качестве примера приведем следующий код программы:

```
Var S : String;
Begin
```

```
  S:='Маша ела кашу';
  Delete(S, 5, 4);
  WriteLn(S);
End.
```

который выведет на экран строку 'Маша кашу'.

Многомерные массивы

В предыдущем модуле мы научились работе с массивами, доступ к элементам которых осуществляется посредством всего одного индекса. Однако, зачастую, приходится хранить данные, доступ которым должен осуществляться не только по одному индексу, а по нескольким. Например, хранение значений некоторой таблицы. В этом случае, в качестве индексов будет выступать номер строки и номер столбца.
С точки зрения синтаксиса, ничего нового при объявлении и использовании многомерных массивов нет. Вспомним как мы объявляли обычные одномерные массивы:

```
Var Имя_массива: Array[Диапазон_Индексов] Of Тип_данных;
```

но ведь массив – это тоже тип данных, поэтому нам никто не запретит объявить переменную, которая является массивом массивов:

```
Var Имя_массива: Array[Диапазон_Индексов] Of
  Array[Диапазон_Индексов] Of Тип_Данных;
```

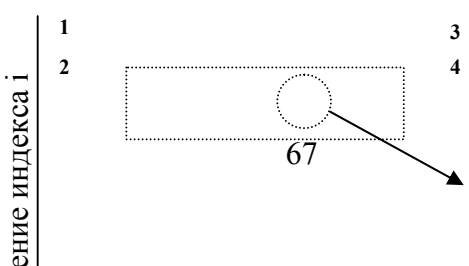
Тем самым, мы и получим двумерный массив (два индекса, значит – два измерения, поэтому и двумерный).
Рекомендованный нами способ объявления массивов можно применить и в данном случае, тогда запись будет более длинной (что с лихвой компенсируется дальнейшим удобством работы с объявленными таким образом массивами). Для двумерного массива такой способ объявления может иметь следующий вид:

```
Const
  ROWS=10;
  COLS=5;
Type
  TRow=Array[1..COLS] Of Integer;
  TTable=Array[1..ROWS] Of TRow;
Var
  Table: TTable;
```

Работа с такими массивами тоже довольна ясна. Доступ к каждому элементу массива осуществляется при помощи двух индексов и записывается в виде:

```
Table[i][j]
```

где `i` и `j` – некоторые переменные, находящиеся в допустимом диапазоне индексов (`i` должна изменяться от `1` до `ROWS`, `j` – от `1` до `COLS`). Такая запись не является случайной. Ведь, как мы выяснили, `Table` является массивом массивов. Поэтому, первым индексом мы получаем доступ к `i`-ому элементу массива, который, в свою очередь, является массивом, к элементам которого мы получаем доступ при помощи индекса `j`.



3-ий элемент исходного многомерного массива.

Ввод и вывод значений многомерного массива

Для ввода или генерации элементов массива необходимо их последовательно перебирать, присваивая или вводя соответствующие элементы. Для этого необходимо использовать столько вложенных циклов, какова размерность массива. Так, для двумерного массива потребуется два цикла для перебора всех его элементов во всех двух измерениях:

- Ввод значений с клавиатуры

```
For i:=1 To ROWS Do
     For j:=1 To COLS Do
          Read(Table[i][j]);
```

- Генерация значений при помощи генератора псевдослучайных чисел

```
Randomize;
For i:=1 To ROWS Do
     For j:=1 To COLS Do
          Table[i][j]:=Random(100);
```

Для того, чтобы красиво вывести двумерный массив на экран, необходимо использовать форматирование при выводе, иначе столбцы чисел будут «прыгать» и выводиться неровно. Кроме того, после каждой строчки необходим переход на новую строку, это можно сделать при помощи пустого `WriteLn`:

```
For i:=1 To ROWS Do
Begin
  For j:=1 To Cols Do
       Write(Table[i][j]:5);
  WriteLn;
End;
```

Изменение порядка строк в таблице

Используя рекомендованный нами способ объявления массивов, применительно к двумерным массивам, покажем, что задача по перестановке строк в таблице принимает очень простой и естественный вид. Это возможно, благодаря тому, что при указанном способе объявления, целое измерение двумерного массива (строчка), выделена в отдельный тип данных, поэтому, можно объявить переменные этого типа данных и оперировать ими как с обычными скалярным (т.е. не массивами) переменными.
Итак, у нас будет таблица, состоящая из целых чисел размером M строчек и N столбцов. Наша задача вывести на экран исходный вид таблицы, поменять местами строчки и вывести на экран модифицированный вид таблицы. Элементы массива, представляющие собой ячейки таблицы, будет генерировать при помощи генератора псевдослучайных чисел.

```
Uses Crt;
```

```
Const
  N=5;
  M=5;
Type
  TRow=Array[1..N] Of Integer;
  TTable=Array[1..M] Of TRow;
Var
  Table : TTable;
  SwapRow: TRow;
  i, j : Integer;
Begin
ClrScr;
Randomize;
For i:=1 To M Do
Begin
  For j:=1 To N Do
  Begin
       Table[i][j]:=Random(50); {Генерируем значения}
       Write(Table[i][j]:5);     {тут же выводим на экран}
  End;
  WriteLn;  {переход на следующую строчку}
End;
For i:=1 To M Div 2 Do {Меняем порядок строк}
Begin
  SwapRow:=Table[i];
  Table[i]:=Table[M-i+1];
  Table[M-i+1]:=SwapRow;
End;
{Выводим на экран измененную таблицу}
WriteLn('---------------');
For i:=1 To M Do
Begin
  For j:=1 To N Do
       Write(Table[i][j]:5);
  WriteLn;
End;
End.
```

В этом коде мы использовали переменную `SwapRow` для хранения целой строчки таблицы при изменения порядка строк. Она имеет тип данных `TRow` – такой же, как и все строки таблицы, поэтому присваивание `SwapRow:=Table[i]` является корректным.

Практикум: Разработка программы, шифрующей тексты

Разработаем программу, которая будет зашифровывать введенный пользователем текст при помощи специального слова – ключа. Расшифровать такой текст можно только в случае, если этот ключ будет известен. В противном случае – восстановление исходного текста будет довольно затруднительным.

Как правило, длина ключа меньше чем длина шифруемого текста, поэтому ключ будет повторяться по длине исходного текста, например, если мы хотим зашифровать фразу «Маша ела кашу» при помощи ключа «ключ», то получится следующая таблица соответствия символов ключа и исходного текста:

М	**а**	**ш**	**а**		**е**	**л**	**а**		**к**	**а**	**ш**	**у**
к	**л**	**ю**	**ч**	**к**	**л**	**ю**	**ч**	**к**	**л**	**ю**	**ч**	**к**

Таким образом, мы видим, что один и тот же символ будет шифроваться разными символами ключа, например, в первом случае, символ «а» шифруется символом «л» ключа, а во втором – «ч». Это приведет к тому, что в результирующем зашифрованном тексте не будет возможности, проведя статистический анализ встречаемости символов, восстановить исходный текст (как это, например, сделал Шерлок Холмс в романа Конан Дойля «Пляшущие человечки»), обладая информацией о частоте появления различных символов в текстах (почти в любых языках самый часто встречающийся символ в тексте – это пробел).

Шифрование и расшифровка будут основываться на логической функции XOR, таблицу истинности которой мы приведем ниже

A	B	A Xor B
1	**1**	**0**
1	**0**	**1**
0	**1**	**1**
0	**0**	**0**

Т.е., в отличие от AND и OR, оператор XOR будет истинен только в том случае, если оба операнда различны по своему значению.

Как оператор XOR можно применить при шифровании не битов (как это приведено в таблице), а для символов? Как мы знаем, каждый символ имеет свой код, а код, в свою очередь, можно представить как последовательность битов в двоичной системе исчисления.

Например, символ ‘s’ имеет код 115, который в двоичной системе записывается в виде 1110011. Если, допустим, получилось так, что этот символ придется шифровать при помощи символа ‘d’ ключа, имеющим код 100 или в двоичной системе – 1100100, то применяя вышеприведенную таблицу, получим

‘s’	**1**	**1**	**1**	**0**	**0**	**1**	**1**
				Xor			
‘d’	**1**	**1**	**0**	**0**	**1**	**0**	**0**
#23	**0**	**0**	**1**	**0**	**1**	**1**	**1**

Т.е. в результате шифрования этих двух символов, получился символ, с кодом 23. Этот символ является так называемым «непечатным» символом и является служебным. Но сохранить его, например, в файл, можно.

Расшифровка закодированного текста осуществляется по такому же алгоритму, только вместо шифруемого символа подставляется символ закодированной строки. В нашем примере это выглядит следующим образом:

#23	**0**	**0**	**1**	**0**	**1**	**1**	**1**
				Xor			
‘d’	**1**	**1**	**0**	**0**	**1**	**0**	**0**
‘s’	**1**	**1**	**1**	**0**	**0**	**1**	**1**

Как мы видим, применив предложенный алгоритм в обратную сторону, мы получили исходный символ ‘s’.

Для реализации программы, шифрующей тексты, таким образом, нам потребуются две функции:

1. `Ord` – возвращает по символу его код;
2. `Chr` – возвращает по коду соответствующий ему символ.

```
Uses Crt;
Var
  SourceStr, Key, EncodedStr: String;
  i, j : Integer;
Begin
```

```
  ClrScr;
  WriteLn('Введите исходную строку');
  ReadLn(SourceStr);
  WriteLn('Введите ключ');
  ReadLn(Key);
  EncodedStr:=SourceStr;
  j:=1;
  For i:=1 To Length(SourceStr) Do
  Begin
       EncodedStr[i]:=Chr(Ord(SourceStr[i]) Xor Ord(Key[j]));
       {реализуем цикличность ключа}
       If j=Length(Key) Then j:=j+1 Else j:=1;
  End;
  WriteLn('Зашифрованный текст: ', EncodedStr);
End.
```

Модуль 6. Записи и множества

Пользовательские типы данных и записи

До сих пор мы работали с массивами, которые хранили лишь один какой-то срез необходимых нам значений: мы могли хранить в массиве какие-то значения, но к чему относятся эти значения мы определить не могли. Например, если наша задача заключается в поиске самого успешного ученика в классе, то мы можем, храня средние оценки всех учеников в одном массиве, найти максимальное значение (которое и будет соответствовать наилучшей успеваемости), однако у нас нету средств определения какому школьнику эта успеваемость принадлежит. Это проблему можно, конечно, решить, заведя второй массив, у которого на соответствующих оценкам позициях будут располагаться фамилии учеников. Определив индекс максимального элемента, мы выведем фамилию, располагающуюся во втором массиве на том же месте, что и максимальная оценка. Грубо говоря у нас могла быть такая схема

Массив 1 – средние оценки

Значение	4.2	3.5	4.5	5	4.3
Индекс	1	2	3	4	5

Массив 2 – фамилии

Значение	Иванов	Петров	Сидоров	Сергеев	Васильев
Индекс	1	2	3	4	5

Использовав алгоритм поиска максимального значения в первом массиве, мы определим, что номер максимального элемента будет равен 4, а взяв из второго массива фамилию, располагающейся на том же четвертом месте, мы сможем определить, что фамилия наиболее успевающего ученика – Сергеев.

Такой подход возможен, но не очень удобен. Поэтому для объединения разнотипных данных в одну единую структуру (в данном случае у нас есть необходимость хранить в одном месте оценку (это вещественное число) и фамилию (это строка)) используются специальные типы данных – записи.

Запись (record) – это специальный тип данных в Pascal, позволяющий объединять в себе разнотипные поля и оперировать ими как единым целым. Это своеобразный контейнер, хранящий в себе переменные любого типа данных.

Прежде чем использовать запись, ее структуру необходимо описать в специальном разделе блока деклараций, который начинается с ключевого слова `Type`. Для этого необходимо дать имя записи (оно подчиняется правилам именования идентификаторов) и перечислить все поля и типы данных, которые входят в нее. Общий синтаксис выглядит следующим образом:

```
Type ИМЯ_ЗАПИСИ=Record
  Имя_Поля_1 : Тип_данных;
  Имя_Поля_2 : Тип_данных;
  …
  Имя_Поля_N : Тип_данных;
End;
```

Тем самым, с точки зрения Pascal, мы создали новый тип данных, который можно использовать при объявлении переменных.

В случае, если мы объявим переменную, которая имеет тип данных запись, то доступ к полям записи осуществляется через точку следующим образом

```
Var ПЕРЕМЕННАЯ: ИМЯ_ЗАПИСИ;
…
ПЕРЕМЕННАЯ.Имя_Поля_1:=Значение;
```

Например, если мы создадим следующий тип данных

```
Type TStudentMark=Record
```

```
  AvgMark : Real;
  StudentName : String;
End;
```

и объявим переменную этого типа данных

```
Var StudentMark : TStudentMark;
```

то записать значение среднего балла студента и его фамилию можно следующим образом:

```
StudentMark.AvgMark:=4.5;
StudentMark.StudentName:='Иванов';
```

Модуль 7. Функции и процедуры

Прежде чем разбираться в том, что такое *функции* и *процедуры*, давайте рассмотрим следующий участок кода. В нем пользователь должен ввести две даты (день, месяц и год), программа проверит правильность ввода значений (как корректность вводимых значений, так и корректность с точки зрения правильности даты, например 30 февраля – это некорректная дата) и затем эти даты использует для каких-либо целей, например для вычисления разницы между этими двумя датами. Правильность ввода чисел мы будем осуществлять в соответствии с программой, рассмотренной в предыдущих главах. Запаситесь терпением, код хоть и большой, но простой.

```
Var FDay, FMonth, FYear, TDay, TMonth, TYear: Integer;
  Sd, Sm, Sy: String;
  Code: Integer;
  MaxD:Integer;
Begin
  WriteLn('Введите день, месяц, год первой даты');
  ReadLn(Sd, Sm, Sy);
  {Проверяем корректность чисел}
  Val(Sd, FDay, Code);
  If Code<>0 Then
  Begin
       WriteLn('Некорректный день');
       halt;
  End;
  Val(Sm, FMonth, Code);
  If Code<>0 Then
  Begin
       WriteLn('Некорректный месяц');
       halt;
  End;
  Val(Sy, FYear, Code);
  If Code<>0 Then
  Begin
       WriteLn('Некорректный год');
       halt;
  End;
  {Проверяем корректность даты}
  If (FMonth<=0) or (FMonth>12) Then
       Begin
            WriteLn('Дата некорректна');
            halt;
       End;
  case FMonth of
       1,3,5,7,8,10,12: MaxD:=31;
       4,6,9,11: MaxD:=30;
       2: If (FYear mod 4=0) and
            (FYear mod 400<>0) or (FYear mod 400=0) Then
            MaxD:=29 {Год високосный}
       else
            MaxD:=28;
  end;
```

```
  If  (FDay<=0) or (FDay>MaxD) Then
     Begin
       Writeln('Дата Некорректна');
       halt;
      End;
  {Вводим вторую дату}
  WriteLn('Введите день, месяц, год второй даты');
  ReadLn(Sd, Sm, Sy);
  {Проверяем корректность ввода чисел}
  Val(Sd, TDay, Code);
  If Code<>0 Then
  Begin
       WriteLn('Некорректный день');
       halt;
  End;
  Val(Sm, TMonth, Code);
  If Code<>0 Then
  Begin
       WriteLn('Некорректный месяц');
       halt;
  End;
  Val(Sy, TYear, Code);
  If Code<>0 Then
  Begin
       WriteLn('Некорректный год');
       halt;
  End;
  {Проверяем корректность даты… Снова ;-) }
  If (TMonth<=0) or (TMonth>12) Then
       Begin
            WriteLn('Дата некорректна');
            halt;
       End;
  case TMonth of
       1,3,5,7,8,10,12: MaxD:=31;
       4,6,9,11: MaxD:=30;
       2: If (TYear mod 4=0) and
            (TYear mod 400<>0) or (TYear mod 400=0) Then
            MaxD:=29 {Год високосный}
       else
            MaxD:=28;
  end;
  If (TDay<=0) or (TDay>MaxD) Then
  Begin
       Writeln('Дата Некорректна');
       halt;
  End;
{Дальше идет код вычисления разницы между датами….}
```

Как вы видите, одни и те же участки кода повторяются для разных значений переменных. Так, для проверки правильности ввода числе мы использовали следующую конструкцию:

```
Val(Строковая Переменная, Целочисленная переменная, Code);
```

```
If Code<>0 Then
Begin
  WriteLn(Здесь сообщение об ошибке);
  halt;
End;
```

Конечно, в больших программах, такой повторяющийся код может встречаться намного чаще. Разумеется, повторяемость одного и того же участка программы не может быть хорошим знаком. Во-первых, программа становится больше и, следовательно, более требовательной к ресурсам, во-вторых, для внесения изменений в повторяемый участок, потребуется внести изменения в нескольких участках программы, а это может быть источником потенциальных ошибок, так как попросту можно какой-нибудь участок пропустить и/или исправить некорректно.
Поэтому было решено внести в языки программирования конструкции, которые бы избавляли программистов от описанных проблем. Это и есть подпрограммы.
Подпрограммы – это обособленные от основной программы операторы, которые можно многократно вызывать из различных участков кода. Поскольку, как мы видели из приведенного выше примера, код подпрограммы вроде бы одинаков, но некоторые переменные в нем меняются (так, в нашем примере, вместо `Строковая переменная`, должны быть последовательно переменные `Sm`, `Sd`, `Sy`). Поэтому, для того, чтобы для каждой отдельной переменной не писать отдельную подпрограмму, у них есть список *параметров*, которые указываются при их вызове. Эти параметры также зачастую называются *аргументами* подпрограммы.
В Pascal у каждой подпрограммы есть свое имя. Это имя подчиняется всем правилам идентификаторов, которым подчиняются и константы и переменные. Вызов подпрограммы заключается в указании ее имени, а в скобках после имени – список необходимых аргументов через запятую.
Иногда суть подпрограммы заключается в том, что для указанного списка аргументов, она должна каким-то образом вычислить какое-то значение и возвратить его для дальнейшего использования в вызывающий блок программы. В качестве примера можно привести подпрограмму, которая для заданной строки должна вычислять количество слов в предложении. В качестве ее аргумента будет строка, а значением этой подпрограммы будет количество подсчитанных слов. В этом случае в Pascal такая подпрограмма называется *функцией*.
В случае, если возвращать никакого значения не нужно, то тогда используется другой вид подпрограмм – *процедуры*.

Объявление подпрограмм

Для процедур и для функций в структуре программы на Pascal выделено свое место, где они могут располагаться. Подпрограммы могут находиться до раздела деклараций, после него и между двумя разделами деклараций, если это необходимо.
В зависимости от типа подпрограммы, ее объявление будет немножко меняться.

Объявление функций

Синтаксис объявления функции следующий:

```
Function ИмяФункции(Арг1: Тип1; Арг2: Тип2;…): Тип
возвращаемого результата;
Begin
{Тело функции}
ИмяФункции:=Выражение; {таким образом функция возвращает
                        значение}
End;
```

где
ИмяФункции – это имя функции,
Арг1, Арг2 – имена аргументов их может быть сколько угодно,
Тип1, Тип2 – типы данных аргументов,
Тип возвращаемого результата – тип данных результата функции, т.е. значения, которое она возвращает.
Для того, чтобы указать, какое значение функция должно возвратить, т.е. каков ее конечный результат, используется запись ИмяФункции:=Выражение. Где Выражение – это любая переменная, константное значение или действительно выражение, результат которого и будет результатом функции. Разумеется, тип данных результата этого выражения должен совпадать с типом возвращаемого функцией результата.
Приведем несколько примеров объявления функций.

Пример 1.

Функция, определяющая максимальное из двух чисел. В качестве аргументов этой функции будут эти два числа, а в качестве ее значения – максимальное из них.

```
Function max(a, b: Integer): Integer;
Begin
  If a>b Then
       max:=a
  Else
       max:=b;
End;
```

Пример 2.

Функция, проверяющая является ли заданный год високосным или нет. Номер года будет являться аргументом функции, а результатом ее выполнение будет значение True – если год високосный и False в противном случае. Т.е. тип возвращаемого результата будет Boolean.

```
Function IsLeap(Year: Integer): Boolean;
Begin
  IsLeap:=(Year mod 4=0) and (Year mod 100<>0) or
          (Year mod 400=0);
End;
```

Объявление процедур

Поскольку процедура – это подпрограмма, которая не возвращает значение, то помимо изменившегося ключевого слова при объявлении, будет отсутствовать тип данных возвращаемого значения.

```
Procedure Имя_Процедуры(арг1:Тип1;арг2:Тип2;…);
Begin
  {Тело процедуры}
End;
```

Аргументы и их типы указываются таким же образом, как и у функции.
Приведем примеры объявления процедур.

Пример 1.

Процедура, печатающая красным текстом сообщение об ошибке. Текст сообщения передается в процедуру в качестве аргумента. Для успешного выполнения этой процедуры потребуется модуль Crt, так как в нем описана процедура изменения цвета текста.

```
Procedure ErrMsg(StrMsg:String);
Begin
  TextColor(RED);
  WriteLn('Ошибка: ', StrMsg);
End;
```

Вызов подпрограмм

Использование подпрограмм существенно зависит от того, с каким видом мы имеем дело – с функцией или процедурой.
В случае, если необходимо вызвать процедуру, то достаточно указать ее имя и передать все параметры:

```
Procedure ErrMsg(StrMsg:String);
Begin
  TextColor(RED);
  WriteLn('Ошибка: ', StrMsg);
End;

Var a: Integer;
Begin
  ReadLn(a);
  If a<0 Then
       ErrMsg('Число не может быть отрицательным!');
  …
```

Функция, в отличие от процедуры, может возвращать значение, поэтому это значение можно записывать в переменные или передавать в другие функции/процедуры.

```
Function max(a, b: Integer): Integer;
Begin
  If a>b Then
       max:=a
  Else
       max:=b;
End;

Var v1, v1: Integer;
    vm: Integer;
Begin
  Readln(v1, v2);
  vm:=max(v1,v2);
  WriteLn('Максимальное число=', vm);
End.
```

Функцию также можно использовать и таким образом:

```
Function IsLeap(Year: Integer): Boolean;
Begin
```

```
  IsLeap:=(Year mod 4=0) and (Year mod 100<>0) or
          (Year mod 400=0);
End;

Var Y:Integer;
Begin
  Readln(Y);
  If IsLeap(Y)=True Then
       WriteLn('Год високосный')
  Else
       WriteLn('Год невисокосный');
End.
```

Формальные и фактические параметры

Поскольку параметры подпрограмм используются в двух местах: в заголовке функции и при ее вызове, то чтоб не возникло путаницы, введены следующие термины:

- *Формальные параметры* – это те параметры, которые указываются в заголовке подпрограммы и используются внутри ее тела;
- *Фактические параметры* – это те переменные или значения, которые передаются в функцию из вызывающей ее области кода.

Так, в следующем примере, `a` – это формальный параметр, а `b` и `4` – фактические:

```
Procedure Dummy(a: Integer);
Begin
…
End;

Var b: Integer;
Begin
…
Dummy(b);
Dummy(4);
…
End.
```

Локальные и глобальные переменные. Область видимости переменных

Предположим, что в одной из наших подпрограмм необходимо организовать цикл с параметром, например для написания функции факториала. В этом случае, нам потребуется использовать переменную для цикла. В связи с этим возникает вопрос: где объявлять эту переменную? «Видит» ли подпрограмма те переменные, которые объявлены в разделе деклараций?

Ответ на этот вопрос простой: подпрограмма «видит» те переменные, константы и типы данных, которые объявлены до нее. Приведем наглядные примеры:

Пример 1.

Поскольку переменная `i` объявлена до функции `SomeFunc`, то использование этой переменной возможно и внутри функции.

```
Var i: Integer;
```

```
Function SomeFunc(): LongInt;
Begin
…{здесь можно использовать переменную i}
End;
…
```

Пример 2.

В данном примере использование переменной `i` невозможно. Она «невидима» для фукнции `SomeFunc`, так как объявлена после нее.

```
Function SomeFunc(): LongInt;
Begin
…{здесь переменная i не видна}
End;
Var i: Integer;
…
```

Переменные, которые видимы и для всех подпрограмм и для основной программы называются *глобальными переменными*. Так, в примере 1 переменная i – это глобальная переменная.
Но использование глобальных переменных внутри подпрограмм – это плохая практика. Поскольку эти переменные являются по сути «общими» для всех, некоторые подпрограммы могут не ожидать, что переменную, которую они используют, изменит другая подпрограмма. Это может привести к путанице в коде. По этой причине принято подпрограммы делать автономными от внешнего окружения. Всю необходимую информацию необходимо передавать в подпрограммы посредством их аргументов, по возможности, разумеется.
Но, раз мы запрещаем себе использование глобальных переменных, взамен нужно дать что-то другое, чтобы позволило комфортно разрабатывать подпрограммы. И это «что-то» - *локальные переменные*. По сути, подпрограмма – это маленькая программа, со всеми присущими ей разделами. Т.е. раздел деклараций внутри подпрограммы также присутствует и в нем можно объявлять переменные, константы и типы данных, которые будут использоваться только внутри этой подпрограммы.
Так, возвращаясь к примеру с факториалом, функция с локальными переменными будет выглядеть следующим образом:

```
Function factorial(n: Integer): LongInt;
Var i: Integer;
    Res: LongInt;
Begin
  Res:=1;
  For i:=1 to n do
       Res:=Res*i;
  Factorial:=Res;
End;

Var f: Integer;
Begin
  Readln(f);
  WriteLn('Факториал = ', factorial(f));
End.
```

В рассматриваемом примере, переменные `i` и `Res` являются локальными, так как их можно использовать только внутри функции `factorial`.

А что произойдет, если мы объявим глобальную переменную, и добавим в подпрограмму локальную переменную с тем же именем? Будет ли это ошибкой? Нет, не будет. Локальные переменные имеют более высокий приоритет и «перекроют» собой одноименные глобальные переменные. Давайте убедимся в этом на следующем простом примере.

```
Var MyText:String; {Глобальная переменная}
Procedure OutText;
Var MyText:String; {Локальная переменная}
Begin
  MyText:='В процедуре';
  WriteLn(MyText);
End;

Begin
  MyText:='В Главной программе';
  OutText;
  WriteLn(MyText);
End.
```

Если бы локальная переменная не перекрывала собой глобальную, то в результате выполнения этой программы, было бы два раза написано:

```
  В процедуре
  В процедуре
```

Поскольку в этом случае, процедура `OutText` изменила бы значение глобальной переменной `MyText`. Однако, этого не происходит, мы видим, что внутри процедуры используется локальная переменная `MyText`, а в основной программе, поскольку локальная переменная не видна, используется глобальная переменная.

Рекурсия

Поскольку одну подпрограмму можно вызывать из другой подпрограммы, то никто не запрещает вызвать подпрограмму из самой себя. Например:

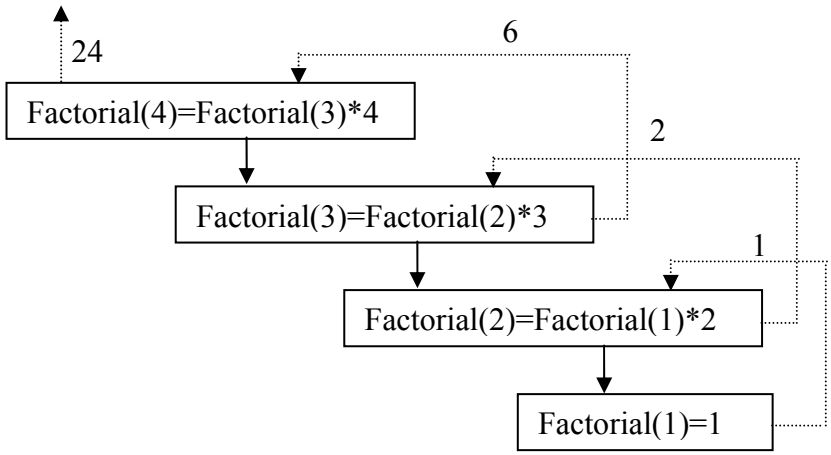
```
Procedure DoIt(a: Byte)
Begin
…
  DoIt(LocalVar); {Рекурсивный вызов}
…
End;
```

Явление, когда подпрограмма вызывает сама себя, называется *рекурсией*.
Рекурсия бывает простой и сложной. Пример простой рекурсии мы привели – это непосредственный вызов подпрограммы из самой себя. Сложная рекурсия – это опосредованный вызов подпрограммы через цепочку других подпрограмм. Например, подпрограмма N вызывает подпрограмму M, а та в свою очередь, вызывает N.
Рекурсию применяют для написания алгоритмов, которые описываются рекуррентными соотношениями. Типичным примером таких алгоритмов является вычисление факториала. Напомним, что факториал описывается рекуррентным соотношением:

$$N! = (N-1)! \cdot N$$

Т.е., зная значение факториала для предыдущего числа, мы можем вычислить факториал нужного нам числа. В связи с этим, можно построить следующую цепочку вычислений, которую мы приведем на примере вычисления факториала 4.

Итак, пусть `Factorial(n)` – это функция вычисления факториала числа `n`. Тогда, учитывая, что нам «известен» факториал 1 – это 1, можно построить следующую цепочку:



Но, если бы у нас не было терминального условия, что при `n=1` функция `Factorial` должна возвратить 1, то такая теоретически цепочка никогда бы не завершилась, и это могло ошибкой *Call Stack Overflow* – переполнение стека вызова. Чтобы понять что такое стек вызова и как он может переполниться, давайте посмотрим на рекурсивную реализацию нашей функции:

```
Function factorial(n: Integer): LongInt;
Begin
  If n=1 Then
       Factorial:=1
  Else
       Factorial:=Factorial(n-1)*n;
End;
```

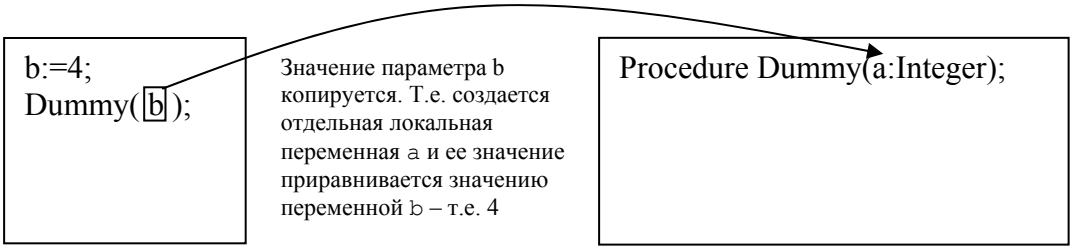
Как мы видим, для того, чтобы цепочка работала корректно, необходимо перед каждым очередным вызовом функции самой себя, где-то сохранять все локальные переменные, чтобы затем при обратном «разворачивании» цепочки результат был правильным (вычисленное значение факториала от `n-1` умножилось на `n`). В нашем случае – при каждом вызове функции `factorial` из самой себя, должны сохраняться все значения переменной `n`. Область, в которой сохраняются локальные переменные функции при рекурсивном обращении к самой себе, называется стеком вызова. Разумеется, этот стек не бесконечный и при неправильном построении рекурсивных вызовов может быть исчерпан. Конечность итераций нашего примера гарантируется тем, что при `n=1` вызов функции прекратится.

Передача параметров по значению и по ссылке

До сих пор мы не могли изменить в подпрограмме значение фактического параметра (т.е. того параметра, который указывается при вызове подпрограммы), а в некоторых прикладных задачах это было бы удобным. Вспомним процедуру `Val`, которая изменяет значение сразу двух ее фактических параметров: первый – это тот параметр, куда будет записано преобразованное значение строковой переменной, а второй – это параметр `Code`, куда помещается номер ошибочного символа, в случае неудачи при преобразовании типа. Т.е. все-таки существует механизм, при помощи которого подпрограмма может изменять фактические параметры. Это возможно благодаря различным способам передачи параметров. Давайте разберемся детально в этих способах.

Передача параметров по значению

По сути, именно таким образом мы передавали все параметры в наши подпрограммы. Механизм следующий: при указании фактического параметра, его значение копировалось в область памяти, где располагается подпрограмма и затем, после того, как функция или процедура завершила свою работу, эта область очищается. Грубо говоря, во время работы подпрограммы, существует две копии ее параметров: одна в области видимости вызывающей программы, а вторая – в области видимости функции.



При таком способе передачи параметров требуется больше времени на вызов подпрограммы, так как помимо самого вызова, необходимо скопировать все значения всех фактических параметров. В случае, если в подпрограмму передается большой объем данных (например, массив с большим количеством элементов), время, требуемое на копирование данных в локальную область, может быть существенным и это необходимо учитывать при разработке программ и поиске узких мест в их производительности.
При таком способе передачи, фактические параметры не могут быть изменены подпрограммой, так как изменения коснутся только изолированной локальной области, которая высвободится после завершения работы функции или процедуры.

Передача параметров по ссылке

При таком способе, значения фактических параметров не копируются в подпрограмму, а передаются адреса в памяти (ссылки на переменные), по которым они располагаются. В этом случае, подпрограмма уже изменяет значения, находящиеся не в локальной области, поэтому все изменения будут видны и вызывающей программе.
Для того, чтобы указать, что какой-либо аргумент необходимо передать по ссылке, перед его объявлением добавляется ключевое слово **var**:

```
Procedure getTwoRandom(var n1, n2:Integer; range: Integer);
Begin
  Randomize;
  n1:=random(range);
  n2:=random(range);
end;

var rand1, rand2: Integer;
Begin
  getTwoRandom(rand1,rand2,10);
  WriteLn(rand1);
  WriteLn(rand2);
End.
```

В этом примере, в процедуру `getTwoRandom` в качестве фактических параметров передаются ссылки на две переменные: `rand1` и `rand2`. Третий фактический параметр (10) передается по значению. Процедура записывает посредством формальных параметров

n1 и n2 два случайных числа в область памяти переменных rand1 и rand2 соответственно.
Способ передачи параметров по ссылки дает прирост в производительности, в случае, если в подпрограмму необходимо передавать большой объем данных. При этом, значения фактических параметров не копируются, а подпрограмме передается лишь одно число – адрес памяти, по которому располагаются данные.
При этом важно помнить, что при использовании механизма передачи параметров по ссылке, нельзя использовать в качестве фактических параметров константы (так как константы нельзя изменять, а механизм подразумевает изменение). Т.е. приведенный ниже пример будет неправильный:

```
Procedure Dummy(var param: Integer);
Begin
…
End;

Begin
   Dummy(5); {Так при передаче параметров по ссылке делать
              нельзя!}
End.
```

Передача массивов в подпрограммы, открытые массивы

Для того, чтобы передать массив в качестве параметра подпрограммы, необходимо объявлять массивы тем способом, который мы советовали в соответствующей главе. Например так:

```
Type TArray=array[1..10] of Integer;
Procedure Dummy(arr:TArr);
Begin
…
End;

Var a:TArray;
Begin
…
Dummy(a);
..
End.
```

Однако, такой способ передачи массив не является универсальным. Он сильно зависит от размера передаваемого массива (размер ведь указан в типе). Зачастую, многие подпрограммы должны быть написаны таким образом, чтобы они могли обрабатывать массивы произвольной длины. Например, если мы хотим написать программу, подсчитывающую среднее арифметическое элементов массива, то нас может не устроить частная реализация только для массивов длиной 10, например.
В связи с этим в Pascal введены так называемые *открытые массивы* – это такое описание параметра в заголовке подпрограммы, в котором не указывается размер массива. Но тогда потребуется определять максимальные и минимальные индексы массива, чтобы универсальным образом их обрабатывать. В этом нам помогут две функции:

1. Low – возвращает минимальный индекс массива;
2. High – возвращает максимальный индекс массива.

Рассмотрим пример:

```
Function Avg(Arr:Array of Integer): Real;
Var i:Integer;
    Sum:Real;
Begin
Sum:=0;
For i:=Low(Arr) To High(Arr) do
  Sum:=Sum+Arr[i];
Avg:=Sum/(High(Arr)-Low(Arr)+1);
End;

Var A:Array[1..10] of Integer;
    B:Array[0..19] of Integer;
    I:Integer;
Begin
Randomize;
For i:=1 to 10 do
  A[i]:=random(50);
For i:=0 to 19 do
  B[i]:=random(100);
WriteLn('Среднее первого массива=', Avg(A));
WriteLn('Среднее второго массива=', Avg(B));
End.
```

В данном примере в функции, вычисляющей среднее арифметическое элементов массива, используются функции Low и High для перебора всех индексов передаваемого в качестве параметра массива. Как мы видим и начальные и конечные индексы у массивов A и B, которые затем предаются в функцию, различаются, но, тем не менее, функция корректно считает среднее арифметическое их элементов. Подсчет количества элементов массива ведется по формуле `Low-High+1`. Где `Low` – это нижний индекс массива, а `High` – верхний.

Модуль 8. Работа с файлами

Ввод с клавиатуры и вывод значений на экран – это не единственные способы ввода и вывода информации для программы. Она также может считывать и записывать данные из файла. Но, если ввод и вывод информации для нас не зависел от типа данных (т.е. мы одинаково выводили строки и числа при помощи функции WriteLn), то для чтения и записи данных из файла нам потребуется указать, с каким именно файлом мы имеем дело. Для этого введен специальный тип данных – *файловая переменная*. Синтаксис ее объявления следующий:

```
Var
  Имя: file of Тип_данных;
```

Например, в таком разделе деклараций

```
Var f: File of Integer;
```

объявляется файловая переменная для работы с файлом, целиком состоящим из целых чисел `Integer`.

С точки зрения Pascal, файлы бывают:

- *типизированные* – тип данных хранящихся в файле явно указывается при объявлении файловой переменной;
- *нетипизированные* – в этом случае файловая переменная объявляется без указания типа данных:

```
Var f: file;
```

- *текстовые* – используются для чтения и записи текстовых файлов. При объявлении файловых переменных, соответствующих текстовым файлам, ключевое слово `file` не используется, вместо него применяется `text`:

```
Var f: text;
```

Независимо от того, какие действия планируется выполнять с файлами, необходимо осуществить следующие операции:

1. Связывание файловой переменной с файлом;
2. Инициализация файловой переменной;
3. Операции с файлом (чтение/запись/дозапись);
4. Закрытие файла.

Связывание файловой переменной с файлом

После того, как файловая переменная определена, необходимо выполнить так называемое *связывание* файловой переменной и конкретного файла. Эта операция осуществляется при помощи процедуры `assign`:

```
Assign(файловая переменная, 'путь к файлу');
```

Например:

```
Assign(f, 'C:\numbers.dat');
```

Операцию связывания необходимо выполнять всегда, независимо от того, какие именно действия планируется совершать с файлов: чтение, запись или дозапись.
Дальнейшие шаги во много зависят от наших планов на файл.

Чтение из файла

При чтении из файла, необходимо выполнить инициализацию файловой переменной, во время которой проверяется существование файла (если файл будет не найден, то программа аварийно завершит свою работу), если таковой найден – то открытие его и перемещение логического указателя на начало файла.

```
Reset(файловая переменная);
```

Для чтения очередной порции данных из файла используется знакомая нам уже процедура `Read`, только список ее аргументов будет расширенным. В качестве первого аргумента используется файловая переменная, по которой определяется из какого файла необходимо производить чтение. В качестве последующих параметров необходимо указывать переменные, куда будут записываться считанные из файла значения. Типы данных этих переменных должны совпадать с типом, указанным при объявлении файловой переменной. При каждом вызове процедуры `Read`, указатель в файле смещается к следующей записи. Поэтому, для того, чтобы считать весь файл целиком, необходимо вызвать процедуру чтения столько раз, сколько записей находится в файле.
Приведем несколько примеров.

Пример 1.

```
Var f: file of Integer;
    Buf: Integer;
Begin
Assign(f, 'c:\datfile.dat');
Reset(f);
Read(f, Buf);
…
```

В данном примере в переменную `buf` будет записано первое число, записанное в файл `c:\datfile.dat`. Если бы потребовалось считать следующее значение, мы бы вызвали процедуру `read` повторно.

Пример 2.

```
Const ROWS=10;
      COLS=10;
Type TRow=array[1..COLS] of Integer;
     TMatrix=array[1..ROWS] of TRow;
Var f: file of TMatrix;
    M:TMatrix;
Begin
Assign(f, 'c:\table.dat');
Reset(f);
Read(f, M);
…
```

В этом примере в переменную M целиком записывается матрица из файла.
Соответственно, записью в этом файле будет считаться таблица из 10x10 целых чисел – т.е. 10x10x2=200 байт (2 байта занимает число Integer, таких чисел 100).

Проверка существования файла

Как мы уже выяснили, при инициализации файла при помощи процедуры `reset`, важно, чтобы файл уже существовал, ведь в противном случае, программа аварийно завершит свою работу, или, проще говоря – «вылетит». Для того, чтобы программа вела себя корректно, необходимо проверить существование файла, и в случае, если файл не будет найден, выдать сообщение пользователю и предпринять какие-либо действия, которые обусловлены конкретной задачей, например – попросить ввести имя пользователя еще раз или корректно завершить работу программы.
Общий алгоритм проверки существования файла будет следующим:

1. Отключается контроль ошибок ввода/вывода при помощи директивы компилятора `{$I-}`;
2. Выполняется инициализация файла процедурой Reset;
3. Контроль ошибок ввода/вывода включается вновь `{$I+}`;
4. Анализируется значение функции `IOResult`. Если эта функция вернула нулевое значение – значит все хорошо, файл существует и к нему открыт доступ, в противном случае – она возвращает ненулевое значение.

Собирая все вместе, получим следующий код:

```
Var filename:string;
  f:file of Integer;
  i:Integer;
Begin
Repeat
  Writeln('Введите имя файла');
  Readln(fileName);
  Assign(f, filename);
  {$I-}
  Reset(F);
  {$I+}
Until IOResult=0;
Read(f, i);
…
```

Чтение всего содержимого файла, функция EOF

Как мы уже заметили, чтение из файла осуществляется по записям. При каждом вызове процедуры `Read` происходит чтение ровно одной записи. Для того, чтобы прочитать файл целиком, необходимо будет выполнить чтение столько раз, сколько записей в файле. В общем случае нам может быть неизвестно количество записей в файле, поэтому часто применяют конструкцию «до тех пор пока не конец файла выполнять чтение», которая реализуется за счет цикла `While` и функции `EOF`.
Функция `EOF` расшифровывается как End Of File – конец файла. Ее аргументом является файловая переменная, ассоциированная с файлом из которого производится чтение. Функция возвращает `True`, если достигнут конец файла и `False` – в противном случае.
Таким образом, схема чтения всего содержимого файла будет следующей:

```
Var f: File Of DataType;
    Buf: DataType;
Begin
Assign(f, ИМЯ_ФАЙЛА);
{$I-}
```

```
Reset(f);
{$I+}
If IOResult<>0 Then
Begin
   WriteLn('Файл не найден');
   halt;
End;
While Not EOF(f) Do
Begin
  Read(f, Buf);
  {Операции с переменной Buf}
  ..
End;
…
```

В приведенном выше участке кода `DataType` – это любой тип данных, который одинаков у переменной `Buf` и у типа данных файловой переменной.

Запись в файл

Инициализация файловой переменной при записи в файл осуществляется при помощи процедуры Rewrite. Работает она следующим образом:

- если файл с заданным именем существует, то он удаляется и вместо него создается новый пустой файл;
- если файла с заданным именем не существует, то он создается.

Таким образом, после выполнения процедуры `rewrite` мы будем иметь дело с пустым файлом, готовым для записи (исключения составляют случаи, когда у нас нет прав доступа к файлу или нет разрешений на запись файла). Синтаксис процедуры следующий:

```
Rewrite(файловая переменная);
```

Непосредственной запись осуществляется при помощи процедуры Write, синтаксис которой, также расширен. Первый ее аргумент – это файловая переменная, ассоциированная с файлом, в который планируется записывать, а все последующие – это переменные, значения которых необходимо записать в файл:

```
Write(файловая переменная, Переменная1, переменная2,…);
```

В случае записи в типизированный файл, тип данных записываемой переменной должен совпадать с типом данных, указанным при объявлении файловой переменной.

Пример 1.

Приведем пример записи в файл матрицы, сгенерированной случайным образом.

```
Const ROWS=10;
      COLS=10;
Type TRow=array[1..COLS] of Integer;
     TMatrix=array[1..ROWS] of TRow;
Var f: file of TMatrix;
    M:TMatrix;
Begin
Assign(f, 'c:\table.dat');
Rewrite(f);
```

```
Write(f, M);
…
```

Закрытие файла

После всех операций чтения/записи в файл, его необходимо закрыть, чтобы высвободить все связанные с ним ресурсы. Для этого предназначена процедура Close, единственным аргументом которой является файловая переменная, ассоциированная с файлом, который необходимо закрыть:

```
Close(f);
```

После вызова этой процедуры, доступ к файлу по файловой переменной невозможен. Для повторного доступа, необходимо заново инициализировать файловую переменную при помощи процедур `Reset` или `Rewrite`.

Текстовые файлы

Текстовые файлы – это особый вид файлов в Pascal, созданный для удобства работы с файлами, содержащими текстовую информацию. Среди удобных особенностей можно отметить следующие:

- поддержка дозаписи (append);
- поддержка процедур WriteLn и ReadLn.

А в остальном, работа с текстовыми файлами не отличается от работы с обычными типизированными файлами. Например, следующий участок кода записывает строчку “Hello, World” в файл:

```
Var txtFile:text;
    S:String;
Begin
Assign(txtFile, 'C:\text.txt');
Rewrite(txtFile);
S:='Hello, World';
WriteLn(txtFile,S);
Close(txtFile);
End.
```

А вот программа, считывающая весь текстовый файл построчно и выводящая его содержимое на экран. Единственное ограничение – длина строки в исходном файле не должна превышать максимальную длину строки в Pascal, т.е. 255 символов. В противном случае такая строка будет выводиться отсеченной.

```
Var f:text;
    s:string;
Begin
Assign(f,'some.txt');
Reset(f);
While not eof(f) do
begin
  Readln(f,s);
  WriteLn(s);
end;
Close(f);
end.
```

Дозапись в текстовый файл

Режим дозаписи – это такой режим, при котором записи добавляются в файл после уже существующих, таким образом сохраняя предыдущее содержимое файла.
Для того, чтобы открыть текстовый файл для дозаписи используется процедура `Append`, вместо `Rewrite`. Аргументы у этой функции такие же – файловая переменная, ассоциированная с необходимым файлом. Эта процедура, проверяет, существует ли файл, если существует – открывает его и перемещает указатель в его конец, в противном случае программа аварийно завершает свою работу.

```
Append(Файловая_переменная);
```

Поскольку эта процедура также проверяет существование файла, то факт его отсутствия также можно отдельно рассмотреть, проанализировав возвращаемое значение функции `IOResult`, также как мы это делали в случае чтения файла (используя директивы `{$I-}`, `{$I+}`).
Все дальнейшие действия аналогичны обычному режиму записи в файл.
Следующая программа, реализует функционал простенького текстового редактора: пользователь вводит имя файла, вводит строку текста и эта строка добавляется в конец существующего файла. Если введенный пользователем файл не существует – то он создается.

```
Var f:text;
    filename,s:string;
Begin
WriteLn('Введите имя файла:');
ReadLn(filename);
Assign(f,filename);
{$I-}
Append(f);
{$I+}
If IOResult<>0 then
begin
   WriteLn('---Файл не найден и будет создан!');
   Rewrite(f);
end;
WriteLn('Введите добавляемую в файл строку');
Readln(S);
WriteLn(f,s);
Close(f);
End.
```

Обратите внимание на то, как в этой программе осуществляется инициализация файловой переменной: сначала делается попытка открыть файл на дозапись, если эта попытка не увенчалась успехом (функция `IOResult` вернула ненулевое значение), то вызывается процедура `Rewrite`, создающая файл.