

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ  
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО  
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ  
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ  
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА»  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

*В.Д. ЕЛЕНЕВ, М.Ю. ГОГОЛЕВ*

**АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ  
НА ЯЗЫКАХ ВЫСОКОГО УРОВНЯ**

*Утверждено Редакционно-издательским советом университета  
в качестве лабораторного практикума*

САМАРА  
Издательство СГАУ  
2010

УДК СГАУ: 004.43(075)  
ББК 32.973я7  
Е504

*Еленев В. Д.*  
Е504 **Алгоритмические языки и технологии программирования на языках  
высокого уровня:** лабораторный практикум / *В.Д. Еленев, М.Ю. Гоголев.* -  
Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2010. - 192 с.

**ISBN 978-5-7883-0748-0**

Рекомендован для студентов высших учебных заведений, обучающихся по направлению 160400.68 «Ракетные комплексы и космонавтика» магистерская программа «Проектирование и конструирование космических мониторинговых и транспортных систем».

Разработан на кафедре летательных аппаратов СГАУ.

УДК СГАУ: 004.43(075)  
ББК 32.973я7

**ISBN 978-5-7883-0748-0**

© Самарский государственный  
аэрокосмический университет, 2010

# Лабораторная работа № 1

## Введение в Object Pascal. Арифметические и логические операторы. Простые типы данных. Преобразование типов.

### Введение

Любые данные, т.е. константы, переменные и выражения, в Object Pascal характеризуются своими типами. Тип определяет множество допустимых значений, которые может иметь тот или иной объект, а также множество допустимых операций применимых к нему. Кроме того, тип определяет также формат внутреннего представления данных в памяти компьютера.

Язык Object Pascal поддерживают строгую типизацию данных. Всем переменным и константам в программе обязательно сопоставлены типы и компилятор строго отслеживает чтобы операции совершаемые над переменными соответствовали их типу. Например, для следующего выражения компилятор сгенерирует ошибку.

```
var x, y, z:integer;
```

```
...
```

```
x:=y/z;
```

Тип результата после вычисления выражения будет вещественным, а переменная в которую должен быть записан результат, имеет целочисленный тип, поэтому компилятор и сгенерирует ошибку.

### Простые типы данных

Все поддерживаемые типы данных можно разделить на две большие группы: простые типы данных и структурированные. Также иногда выделяют некоторые другие типы.

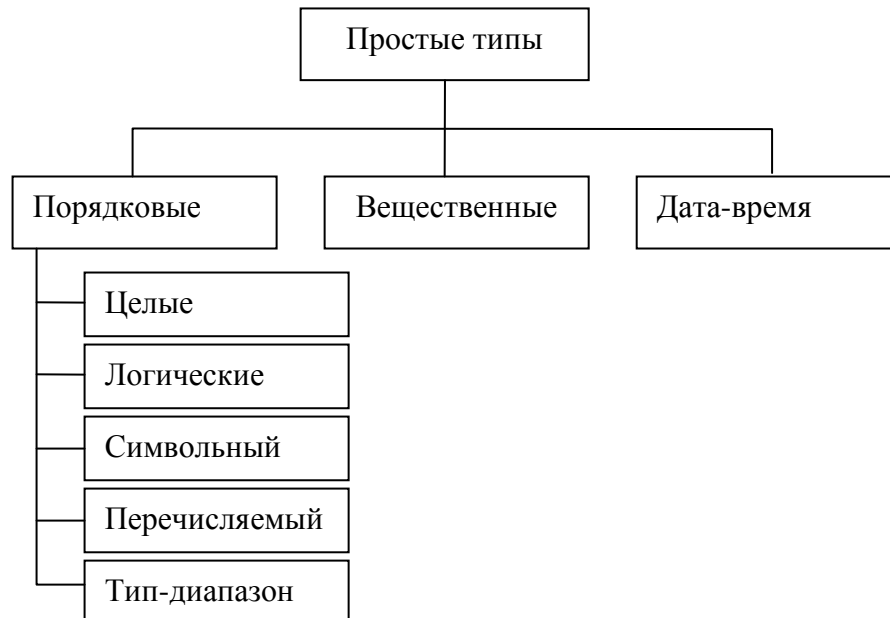


Рисунок 1 - Структура простых типов данных

**Порядковые типы** отличаются тем, что каждый из них имеет конечное число возможных значений. Эти значения можно определенным образом упорядочить и, следовательно, сопоставить некоторое целое число – порядковый номер значения.

**Вещественные типы** предназначены для работы с вещественными числами в формате с плавающей и фиксированной точкой.

**Тип дата-время** предназначен для хранения даты и времени. Фактически для этих целей используется вещественный формат.

## Порядковые типы

Ко всем порядковым типам можно применять следующие функции:

Pred(X) – возвращает предыдущее значение порядкового типа;

Succ(X) – возвращает следующее значение порядкового типа;

High(X) – возвращает максимальное значение порядкового типа;

Low(X) – возвращает минимальное значение порядкового типа.

## Целые типы

Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать один, два, четыре или восемь байт. В таблице 1 приведены основные целочисленные типы.

Таблица 1 – Целочисленные типы

<i>Наименование</i>	<i>Длина, байт</i>	<i>Диапазон значений</i>
Byte	1	0...255
ShortInt	1	-128...+127
SmallInt	2	-32768...+32767
Word	2	0...65535
Integer	4	-2147483648 ... +2147483647
LongInt	4	-2147483648 ... +2147483647
Cardinal	4	0 ... 2147483647
LongWord*	4	0 ... 4294967295
Int64	8	$-2^{63} \dots 2^{63}-1$

Следует учесть, что **Object Pascal** позволяет осуществлять автоматическое приведение типов так там, где в качестве аргумента допустимо использовать переменную типа Word, допускается использовать переменную типа Byte, но не наоборот. Общее правило таково вместо одного типа целочисленной переменной можно использовать другой тип целочисленной переменной их диапазоны значений вкладываются друг в друга. Например возможно преобразование переменной типа Byte в переменную типа Word, но невозможно преобразование ShortInt в Word;

В таблице 2 приведены некоторые функции для работы с целочисленным переменными.

\* Введен в Delphi 6

Таблица 2 – Функции для работы с целыми типами

Функция	Описание
dec(x[, i])	Уменьшает значение X на i, а при отсутствии i на 1.
inc(x[, i])	Увеличивает значение X на i, а при отсутствии i на 1.
odd(x)	Возвращает True, если аргумент – нечетное число.

При действиях с целыми числами тип результата будет соответствовать типу операндов, а если операнды относятся к различным целым типам, – общему типу, который включает в себя оба операнда. Например для типов ShortInt и Word общим типом будет Integer.

При стандартной настройке компилятор не генерирует код проверки осуществляющий контроль за возможным выходом значения из допустимого диапазона, что может привести к неправильной работе программы. Например при выполнении следующей программы на экране появится ноль, а не 65536.

```

program InOutPrg;
{ $APPTYPE CONSOLE }
var x: Word;
begin
  k:=65535;
  k:=k+1;
  writeln(k);
  readln;
end.

```

Если активизировать переключатель Project ⇒ Option ⇒ Compiler ⇒ Range checking и повторить компиляцию, то возникнет ошибка времени исполнения (см. рисунок 2).

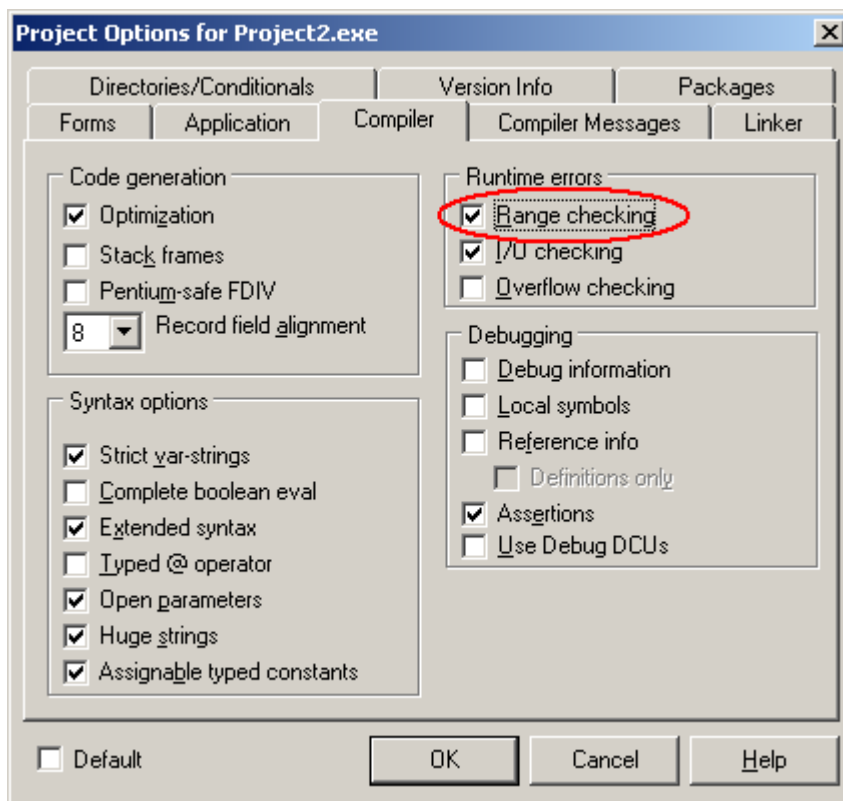



Рисунок 2 – Включение проверки выхода за диапазон


 **Совет:** Для целочисленных переменных безопаснее всего применять тип *Integer*, т.к. диапазон значений этого типа достаточно велик для широкого класса задач. Проверку выхода за диапазон при окончательной компиляции лучше отключить, т.к. она существенно снижает быстродействие программы.

## Логические типы

Переменная логического типа может принимать только два значения True (истина) и False (ложь). Логические типы приведены в таблице 3.

Таблица 3 – Логические типы

<i>Тип</i>	<i>Размер</i>
Boolean	1
ByteBool	1
Bool	2
WordBool	2
LongBool	4

 **Совет:** Для работы с логическими типами используйте тип **Boolean**, остальные типы были добавлены в **Object Pascal** для совместимости с **Windows**.

## Символьный тип

Символьный тип, как следует из его названия, предназначен для хранения кода символа. Внутренняя структура типа совпадает с внутренней структурой беззнаковых целых чисел. Object Pascal поддерживает два символьных типа AnsiChar и WideChar. Характеристики этих типов приведены в таблице 4.

Таблица 4 – Символьные типы

<i>Тип</i>	<i>Размер, байт</i>	<i>Описание</i>
AnsiChar	1	Код символа в кодировке ANSI. Таблица символов этого стандарта состоит из 256 символов, причем первые 128 символов жестко определены стандартом, а остальные 128 могут содержать любые символы. Во вторую часть кодовой таблицы обычно помещают символы национальных алфавитов. Недостатком данного представления символов является то, что невозможно отображение сразу символов более чем 2-х алфавитов.
WideChar	2	Символ в формате UNICOD. Таблица символов содержит 65536 символов. В данной таблице находятся символы всех алфавитов.
Char*	1	Является псевдонимом типа AnsiChar

Для преобразования кода символа в символ следует применять следующую функцию

```
function chr(X:byte):char
```

\* Этот тип является основным символьным типом Object Pascal

Для обратного преобразования используется функция

```
function ord(X:char):byte;
```

Константу символьного типа можно задать двумя способами. Задать символ в одинарных кавычках или указать код символа. Следующие две записи эквивалентны.

```
ch:='A';
```

```
ch:=#97; //97 – код символа “А”.
```

## Перечисляемый тип

Перечисляемый тип задается перечислением тех значений, которые он может получать. Каждое значение именуется некоторым идентификатором и располагается в списке, обрамленном круглыми скобками.

**type**

```
TColors=(red, green, white, yellow);
```

Применение перечисляемых типов делает программы более наглядными и повышает надежность программ. Пусть определены следующие типы:

**type**

```
TColors=(red, green, white, yellow);
```

```
TDays=(monday, tuesday, wednesday);
```

Определены переменные

**var**

```
col:TColors;
```

```
day:TDays;
```

то допустимы следующие операторы

```
col:=white;
```

```
col:=Succ(green);
```

```
day:=Pred(tuesday);
```

но не допустимы

```
col:=monday;
```

```
day:=col;
```

Максимальное количество перечисляемых значений 65536 значений, т.е. во внутреннем представлении перечисляемый тип представляет собой тип Word.

## Тип-диапазон

Тип диапазон представляет собой подмножество некоторого базового порядкового типа. Тип-диапазон задается границами своих значений.

```
<МИН. ЗН.> .. <МАКС. ЗН.>
```

Например

**type**

```
TDigits='0'..'9';
```

```
TMonth=1..12;
```

TMyColors=green .. yellow;

При задании типа-диапазона следует иметь ввиду, что символы “..” нельзя разделять пробелами и что левая граница диапазона не должна превышать правую границу.

## Вещественные типы

Для представления вещественных чисел используются следующие типы приведенные в таблице 5.

Таблица 5 – Вещественные типы Object Pascal

Наименование	Длина, байт	Точность (число значащих чисел)	Диапазон значений
real*	6	11...12	$2,9 \cdot 10^{-39} \dots 1,7 \cdot 10^{38}$
single	4	7...8	$1,5 \cdot 10^{-45} \dots 3,4 \cdot 10^{38}$
double	8	15...16	$5,0 \cdot 10^{-324} \dots 1,7 \cdot 10^{308}$
extended	10	19...20	$3,4 \cdot 10^{-4951} \dots 1,1 \cdot 10^{4932}$
comp	8	19...20	$-2^{63} \dots 2^{63}-1$
currency	8	19...20	$\pm 922\ 337\ 203\ 685\ 477,5907$



**Совет:** Не рекомендуется использовать устаревший тип *real48* в программах, т.к. он разработан для программной эмуляции вещественных чисел и в Object Pascal они сначала конвертируются в *extended*, затем выполняются необходимые операции с вещественными числами после чего происходит обратная конвертация, что приводит к замедлению программы. Для большинства программ наиболее оптимально применение типа *double* (*real*).

## Тип дата-время

Тип дата-время определяется идентификатором *TDateTime* и предназначен для одновременного хранения даты и времени. Во внутреннем представлении переменная этого типа занимает 8 байт. Для работы с данным типом используются функции приведенные в таблице 6\*\*.

Таблица 6 – Подпрограммы для работы с типом *TDateTime*

Подпрограмма	Назначение
<b>function</b> Date: TDateTime;	Возвращает текущую дату
<b>function</b> DateToStr(D: TDateTime):string	Преобразует дату в строку символов
<b>function</b> DateTimeToStr(D: TDateTime):string;	Преобразует дату и время в строку символов
<b>function</b> Time: TDateTime;	Возвращает текущее время

\* Начиная с версии Delphi 5.0 тип *real* соответствует типу *double*, а для совместимости со старыми программами введен тип *real48* соответствующий типу *real* из таблицы 5. При использовании директивы компилятора {*\$REALCOMPATIBILITY ON*} компилятор под типом *real* будет подразумевать именно тип *real48*.

\*\* Для использования функций из таблицы 6 необходимо подключить модуль *SysUtils*. (uses *SysUtils*;) )



**function** TimeToStr (T: TDateTime):string

Преобразует время в строку

## Строковые типы

В Object Pascal определено несколько типов строк (таблица 7).

Таблица 7 – Строковые типы

<i>Тип</i>	<i>Описание</i>
ShortString	Короткая строка. Максимальная длина строки 255 символов. Формат символов AnsiChar. Максимальная длина строки может быть задана, например s: string[20]; строка длиной 20 символов.
AnsiString	Длинная строка. Максимальная длина строки порядка $2^{31}$ символов.
PChar	Строки с завершающим нулем. (используются API Windows). Максимальная длина строки порядка $2^{31}$ символов. Формат символов AnsiChar.
WideString	Широкие строки. Максимальная длина строки порядка $2^{16}$ символов. Формат символа WideChar.
String	По умолчанию является псевдонимом типа AnsiString, если не указана длина строки.

Строковые константы задаются в одинарных кавычках, например

```
var s:string;  
begin  
s:='Hello World!';  
end.
```

Для сцепления двух строк в одну необходимо применять оператор '+', например

```
var s, s1:string;  
begin  
s:='Hello';  
s1:=' World!';  
s:=s+s1;  
writeln(s);  
end.
```

В результате выполнения выше приведенной программы на экран будет выведено Hello World!.

Основные функции для работы со строками приведены в таблице 8.

Таблица 8\* – Процедуры и функции для работы со строками

<i>Подпрограмма</i>	<i>Описание</i>
<b>function</b> Copy (Source: string; Index, Count: Integer):string;	Копирует из строки Source Count символов, начиная с символа с номером Index.
<b>procedure</b> Delete (var Source: string; Index, Count: Integer);	Удаляет Count символов из строки Source, начиная с символа с номером индекс.
<b>procedure</b> Insert (SubSt: string; var St: string; Index: Integer);	Вставляет подстроку SubSt в строку St, начиная с символа с номером Index.
<b>function</b> Length (S:string):integer;	Возвращает текущую длину строки.
<b>function</b> Pos (SubSt, St:string):integer;	Отыскивает в строке St первое вхождение подстроки SubSt и возвращает номер позиции с которой она начинается. Если подстрока не найдена возвращается ноль.

## Преобразование типов

При написании программ часто бывает необходимым преобразовать тип переменной. Близкие типы можно преобразовывать с помощью следующей конструкции.

<идентификатор типа>(<переменная>)

Например преобразование типа переменной Integer к типу Word выглядит следующим образом

```
var i:Integer;
    j:Word;
...
j:=Word(i);
```

В таблице 9 приведены стандартные функции преобразования типов.

Таблица 9 – Стандартные функции преобразования типов

<i>Подпрограмма</i>	<i>Описание</i>
<b>function</b> Trunc(X:real):Integer	Отсекает дробную часть числа
<b>function</b> Round(X:real):Integer	Округляет вещественное число до целого по правилам арифметики
<b>function</b> Val(St:string; var X; Code:Integer)	Преобразует строку символов St в вещественную или целую переменную (в зависимости от типа переменной X). Параметр Code содержит индекс символа который вызвал ошибку при преобразовании или 0 если ошибки не произошло.
<b>procedure</b> Str(X [:Width:[:Decimals]]; var St:string)	Преобразует число X любого вещественного или целого типа в строку символов St. Параметры Width и Decimals,

\* Функции приведенные в этой таблице применимы к типам AnsiString и ShortString.

если они присутствуют, задают общую ширину поля, выделенную под символьное представление числа, и количество символов в дробной части соответственно.

Некоторые функции преобразования типов из модуля **SysUtils** приведены в таблицах 6 и 10.

<i>Подпрограмма</i>	<i>Описание</i>
<b>function</b> StrToInt(s:string):Integer;	Преобразование из строки в целое число
<b>function</b> StrToFloat(s:string):Extended;	Преобразование из строки в вещественное число
<b>function</b> IntToStr(Value:Integer):string;	Преобразование из целого числа в строку
<b>function</b> FloatToStr(Value:Extended):string;	Преобразование из вещественного числа в строку
<b>function</b> FloatToStrF(Value:Extended; Format:TFloatFormat; TOL, Digits):string;	Преобразование из вещественного числа в строку. Параметр Format определяет текстовое представление числа в строке. Более подробная информация по этой функции будет представлена в следующих лабораторных работах.

## Операторы

Все операторы условно можно разделить на три группы: арифметические операторы, логические операторы и специальные операторы.

Арифметические операторы применяются при построении арифметических выражений. К арифметическим операторам относятся +, -, \*, /, (). Результатом вычисления арифметических операторов является целочисленный или вещественный тип.

Пример:

$a+b-c*(d+e)$

Логические операторы применяются при вычислении логических выражений. К логическим операторам относятся >, <, <>, =, and, not, or. Типом результата вычисления логического выражения является логический тип.

Пример

$a>b$

$(a>b)\text{and}(a<c)$

Допускается смешивать логические и арифметические выражения, например

$(a+b)*c=e$

К специальным операторам относится оператор присваивания, операторы ввода-вывода и некоторые другие операторы.

### Оператор присваивания

Оператор присваивания имеет следующий синтаксис

**<идентификатор переменной>:=<выражение>;**

Следует отметить, что между символами := недопустимы пробелы. Тип переменной которой осуществляется присваивание и тип результата вычисления выражения должны совпадать.

## **Оператор SizeOf**

Этот оператор возвращает размер переменной в байтах, он имеет следующий синтаксис

SizeOf(X):Integer;

Здесь X – переменная или имя типа. Оператор возвращает размер переменной заданного типа в байтах.

Пример:

```
var i:Integer;  
    x:real;  
    i:=SizeOf(x);  
    ...  
    i:=SizeOf(Single);
```

## **Операторы ввода-вывода**

Для операций ввода-вывода данных в **Object Pascal** используется пара операторов **read** и **write**. Эти операторы ввода-вывода универсальны и используются при работе с любыми устройствами ввода-вывода такими как, клавиатура, консоль, накопители, порты ввода-вывода и др. Следует также отметить, что эти операторы обрабатываются компилятором особым образом и поэтому идентификаторы **read** и **write** не рекомендуется использовать для именованя собственных объектов.

### **Оператор write**

С помощью оператора **write** осуществляется вывод данных. По умолчанию устройством вывода служит экран. Синтаксис оператора write имеет следующий вид:

write (<выражение>, <выражение>, ... <выражение>);

Выражение может состоять из одной константы или переменной, например

```
write('x=', x);
```

Первое выражение равно 'x=' – это строковая константа. Второе выражение равно x. Это переменная любого простого типа. Следует отметить, что стандартный оператор вывода может работать с переменными только простых типов, а также переменные строкового типа. Оператор **write** после вывода не осуществляет перевод курсора на следующую строку. Для того чтобы после вывода данных курсор переводился на следующую строку необходимо использовать оператор **writeln**.

writeln (<выражение>, <выражение>, ... <выражение>);

Оператор **writeln** без параметров переведет курсор на новую строку.

Операторы **write** и **writeln** позволяют производить форматированный вывод вещественных чисел. Для этого применяется расширенная форма операторов.

writeln (<выражение>:Width:Digits, ...);

write (<выражение>:Width:Digits, ...);

With – определяет общее число символов отведенных на число;

Digits – определяет число выводимых знаков после запятой.

Пример:

```
writeln(x:6:2, ' ', y:6:3);
```

## Оператор read

С помощью оператора **read** осуществляется ввод данных. По умолчанию устройством ввода служит клавиатура. Синтаксис оператора write имеет следующий вид:

```
read (<переменная>, < переменная >, ... < переменная >);
```

Оператор read может работать только с переменными простого типа и строковыми переменными. При вводе нескольких переменных с помощью оператора read они должны отделяться пробелом, а когда все переменные введены необходимо нажать Enter. Если Enter был нажат до ввода всех переменных, то курсор будет переведен на следующую строку и, система будет ожидать ввода оставшихся данных, например

```
program InOutPrg;
{$APPTYPE CONSOLE}
var x, y, z: Integer;
begin
  { TODO -oUser -cConsole Main : Insert code here }
  read(x, y, z);
end.
```

Попробуйте ввести переменные сначала через пробел, а затем используя клавишу ввод для каждой переменной. Как и для оператора write у оператора read существует форма с переводом строки после окончания ввода данных.

```
readln (<переменная>, < переменная >, ... < переменная >);
```

Оператор readln без параметров приостановит выполнение программы до нажатия клавиши Enter.

В ниже приведенном листинге приведен пример программы ввода и вывода данных.

```

program AddPrg;
{ Программа суммирования двух целых чисел }

{$APPTYPE CONSOLE}

var x1, x2, sum:Integer;

begin
write('Enter first number: ');
read(x1);
write('Enter second number: ');
read(x2);
sum:=x1+x2;
writeln('Summa ', x1, 'and', x2, '=', sum);
writeln('Press Enter');
readln;
readln;
end.

```

## Задания к лабораторной работе

1. Напишите программу для вычисления следующих выражений, все переменные целочисленные:

$$a + b * c - (23 + f) * 3$$

$$(a > 3(f + 1)) \text{or} (2f > 3)$$

Ввод исходных данных организуйте с помощью стандартных операторов ввода вывода. При вводе и выводе данных должны присутствовать подсказки.

2. Составьте программу для вывода на экран размеров в байтах всех вещественных типов.
3. Составьте программу для вывода на экран текущей даты и времени.
4. Составьте программу поиска подстроки в строке. Исходные данные строка, подстрока которую следует искать. На экран должен выводиться результат найдена строка или нет (True или False).
5. Составьте программу для перестановки в заданной строке любых двух слов.  
'Delphi is best compiler'
6. Составьте программу для вычисления любого выражения, имеющего вещественный тип результата. Вывод результата на экран должен быть форматированным.

## Вопросы к лабораторной работе

1. Какие простые типы данных вы знаете?
2. Какие целочисленные типы данных вы знаете?
3. Какие типы данных относятся к порядковым, почему?
4. Как задаются символьные константы?
5. Как задать перечисляемый тип данных?

6. Как задать тип диапазон?
7. Какие вещественные типы данных вы знаете?
8. Какие строковые типы данных вы знаете и в чем их отличие?
9. Для чего нужно преобразование типов?
10. Какие функции преобразования вы знаете?
11. Какие операторы Object Pascal вам известны?

## **Справочные таблицы**

Таблица 1 – Целочисленные типы .....	4
Таблица 2 – Функции для работы с целыми типами.....	5
Таблица 3 – Логические типы .....	6
Таблица 4 – Символьные типы .....	6
Таблица 5 – Вещественные типы Object Pascal .....	8
Таблица 6 – Подпрограммы для работы с типом TDateTime.....	8
Таблица 7 – Строковые типы .....	9
Таблица 8 – Процедуры и функции для работы со строками .....	10
Таблица 9 – Стандартные функции преобразования типов .....	10

## Лабораторная работа №2

### Управляющие конструкции языка Object Pascal. Условные и безусловные конструкции. Циклические конструкции.

#### Введение

В лабораторной работе рассматриваются основные управляющие конструкции языка. Это безусловные конструкции, условные конструкции и циклические конструкции. Рассмотрены некоторые директивы компилятора, а также условная компиляция программ.

#### Безусловные конструкции

В Object Pascal существует один оператор безусловного перехода **goto** и четыре функции реализующие безусловный переход (**break**, **continue**, **exit**, **halt**). Назначение всех функций будет рассмотрено ниже.

#### Оператор безусловного перехода Goto

Оператор безусловного перехода предназначен для изменения порядка выполнения операторов в программе. Синтаксис оператора **goto** следующий

**goto** <метка>

**goto** – зарезервированное слово языка Object Pascal;

<метка> – идентификатор или целое число от 0 до 9999. Все используемые метки должны быть объявлены предварительно в разделе **label**.

Листинг 1

```
program GotoPrj;  
  
{$APPTYPE CONSOLE}  
label exit_prg;  
  
begin  
  writeln('Label');  
  goto exit_prg;  
  writeln('Hello world!');  
  exit_prg:  
  readln;  
end.
```

Убедитесь, что текст 'Hello World!' не появится на экране.

#### Функция Halt

Функция Halt осуществляет досрочное завершение программы, т.е. осуществляет безусловный переход на последний оператор программы. Эта функция может быть вызвана в любом месте программы внутри, циклов, условных конструкций и даже функций. Функция Halt имеет следующий синтаксис

Halt (n)

Здесь n – код завершения программы. В последствии данный код может быть проанализирован средствами операционной системы.



## Функция *Exit*

Функция *Exit* завершает работу текущего программного блока. Если она вызывается внутри тела программы, то завершается сама программа. Более подробно данная функция будет описана в следующих лабораторных работах.

## Составной оператор

Составной оператор позволяет представить группу операторов в виде одного оператора. Синтаксис составного оператора следующий

```
begin
    <оператор 1>;
    <оператор 2>;
    ...
    <оператор n>;
end;
```

Пара операторов **begin end** называется операторными скобками. Все операторы заключенные в операторные скобки воспринимаются компилятором как один оператор.

## Условные конструкции

### Конструкция *if ... then ... else*

Конструкция *if then else* имеет следующий синтаксис

```
if <выражение> then <оператор 1> else <оператор 2>;
```

Тип результата вычисления выражения должен быть логическим. Данная конструкция работает следующим образом: если результат вычисления выражения «истина», то выполняется оператор 1, иначе оператор 2. Обратите внимание, что после оператора 1 точка не ставится. Выражение может вырождаться в переменную логического типа. В качестве примера рассмотрим программу для вычисления корней квадратного уравнения.

Листинг 2

```
program SQR_PRG;
{ $APPTYPE CONSOLE }
var a, b, c : real; //коэффициенты уравнения
      d,
      x1, x2 : real; //дискриминант
                        //корни уравнения

begin
  writeln('vvedite koeff');
  readln(a, b, c);

  d:=b*b-4*a*c;

  if d>=0 then
    begin
      x1:=(-b+sqrt(d))/(2*a);
      x2:=(-b-sqrt(d))/(2*a);
      writeln('x1=', x1:6:2);
      writeln('x2=', x2:6:2);
    end
  else writeln('No real roots');
```

```
readln;  
end.
```

Вычисления производятся по следующим известным формулам  $d = b^2 - 4ac$ ,  $x_{1,2} = \frac{-b \pm \sqrt{d}}{2a}$ . Действительные корни уравнения существуют только при условии  $d \geq 0$ .

У этой условной конструкции существует сокращенная форма, ее синтаксис следующий

```
if <выражение> then <оператор 1>;
```

Данная конструкция работает следующим образом: если результат вычисления выражения «истина», то выполняется оператор 1, иначе оператор 1 не выполняется. Для примера рассмотрим программу которая вычисляет сумму покупки в зависимости от стоимости и количества товара, причем при превышении некоторой суммы дается скидка.

Листинг 3

```
program Pokupka;  
  
{ $APPTYPE CONSOLE }  
const discount=0.3; //размер скидки  
      max_summa=500; //максимальная сумма покупки  
  
var n      :integer; //количество товара  
    Price  :real;    //цена единицы товара  
    Summa  :real;    //сумма покупки  
  
begin  
  writeln('vvedite kol-vo tovara');  
  readln(n);  
  writeln('vvedite stoimost'' tovara');  
  readln(Price);  
  
  Summa:=n*Price;  
  
  if Summa>max_Summa then Summa:=Summa*(1-discount);  
  
  writeln('Summa pokupki ', summa:4:2);  
  
  readln;  
end.
```

На месте операторов 1 и 2 могут находиться любые операторы в том числе и условные. В качестве примера изменим программу в листинге 3. Теперь при подсчете учитывается, что сумма скидки не может превышать некоторого максимального значения.


Листинг 4

```
program Pokupka2;  
  
{ $APPTYPE CONSOLE }  
const discount =0.3; //размер скидки  
      max_summa=500; //максимальная сумма покупки  
      max_discount=300; //максимальная сумма скидка  
  
var n      :integer; //количество товара  
    Price  :real;    //цена единицы товара  
    summa  :real;    //сумма покупки  
  
begin  
  writeln('vvedite kol-vo tovara');  
  readln(n);  
  writeln('vvedite stoimost'' tovara');
```

```

readln(Price);
Summa:=n*Price;
if Summa>max_Summa then
  if Summa*discount>max_discount then Summa:=Summa-max_discount
  else Summa:=Summa*(1-discount);
writeln('Summa pokupki ', summa:4:2);
readln;
end.

```

 **Совет:** Не используйте конструкций вида *if b=true then ...* Где *b* переменная логического типа. Вместо этого достаточно написать *if b then ...*.  
Вместо конструкции *if x>2 then b:=true;* следует писать *b:=x>2;* Такой код будет выполняться быстрее, а его запись более компактна.

Если необходимо проверить несколько независимых друг от друга условий, то следует применять булевы операторы **and**, **or**. Например для проверки попадания величины в заданный интервал используется следующее булево выражение  $(a>1)\mathbf{and}(a<2)$ .

В следующем примере производится проверка попадания заданной величины в один из двух заданных интервалов (интервалы не пересекаются).

Листинг 5

```

program Interval;
{$APPTYPE CONSOLE}
const //первый интервал
      x1=1;
      x2=5;
      //второй интервал
      x3=10;
      x4=20;
var x :integer;
begin
  write('vvedite x ');
  readln(x);
  if ((x>=x1)and(x<=x2))or((x>=x3)and(x<=x4)) then writeln('Point
in interval')
  else writeln('Point not in interval');
  readln;
end.

```

### Конструкция *case ... of*

Для осуществления множественного выбора более удобной является конструкция *case ... of*. Она имеет следующий синтаксис

```

case <переменная> of
  <знач. 1>:<оператор 1>;
  <знач. 2>:<оператор 2>;

```

```

...
<знач. n>:<оператор n>;
else
<операторы>
end;

```

Здесь переменная должна быть любого порядкового типа. Алгоритм работы этой конструкции следующий. Значение переменной сравнивается со значениями ключей определенных в разделе **case end**. Если значение переменной равно значению одного из ключей, то выполняется соответствующий оператор. Если ни одно значение не совпало, то выполняются операторы в секции **else**. Секция **else** является необязательной и может отсутствовать.

Модифицируем программу приведенную в листинге 4 таким образом, чтобы в выходные дни действовала дополнительная скидка в размере 2% на всю сумму покупки.

Листинг 6

```

program Pokupka3;

{$APPTYPE CONSOLE}
const discount=0.3; //размер скидки
      max_summa=500; //максимальная сумма покупки
      max_discount=300; //максимальная сумма скидка

var n :integer; //количество товара
     Price:real; //цена единицы товара
     summa:real; //сумма покупки
     day_of_week :integer; //день недели от 1-7
begin
  writeln('vvedite kol-vo tovara');
  readln(n);
  writeln('vvedite stoimost' ' tovara');
  readln(Price);

  Summa:=n*Price;

  if Summa>max_Summa then
    if Summa*discount>max_discount then Summa:=Summa-max_discount
    else Summa:=Summa*(1-discount);

  write('vvedite den nedeli ');
  readln(day_of_week);

  case day_of_week of
    1..5: ;
    6,7 : Summa:=Summa*0.98;
  else
    halt(0);
  end;

  writeln('Summa pokupki ', summa:4:2);

  readln;
end.

```

В качестве значения ключа может выступать диапазон значений или список значений, а также их комбинации, например

```

case n of

```

```

1:          <оператор>;
3, 4:      <оператор>;
5..9:      <оператор>;
10, 12..17: <оператор>;
end;

```

Отметим еще раз, что в качестве переменной в операторе множественного выбора может выступать переменная любого порядкового типа поэтому можно написать, например, следующий код

```

type TLineStyle=(lsSolid, lsDot, lsDash, slDashDot)
var LineStyle:TLineStyle;
...
case LineStyle of
  lsSolid:  <оператор 1>;
  lsDot:    <оператор 2>;
  lsDash:   <оператор 3>;
  lsDashDot: <оператор 4>;
end;
...

```

или следующий

```

var Flag:Boolean;
...
case Flag of
  True:    <оператор 1>;
  False:   <оператор 2>;
end;
...

```

## Циклические конструкции

Циклы позволяют многократно выполнять отдельный оператор. В Object Pascal существует три вида циклических конструкций. Это цикл со счетчиком, цикл с предусловием и цикл с постусловием.

### **Конструкция for ...to ...do**

Первая из конструкций позволяющая организовать повторение операторов называется циклом со счетчиком или перечисляемым циклом. В этом операторе обязательно указываются следующие параметры:

- имя переменной любого порядкового типа в которой хранится число повторений цикла – счетчик цикла.;
- начальное значение для переменной цикла;
- конечное значение для переменной цикла.

Синтаксис конструкции **for to do** имеет следующий вид

```
for <перем._цикла>:=<нач._знач> to (downto) <кон._знач> do <оператор>;
```

При после каждого выполнения переменная увеличивается на единицу, а если заменить зарезервированное слово **to** на **downto** то уменьшаться на единицу. Таким образом в этой конструкции число повторений цикла фиксировано. Переменная цикла

(счетчик цикла) может использоваться в теле цикла, но не может изменяться в теле цикла. Ниже приведенная программа производит вычисление суммы чисел от 1 до n.

Листинг 7

```
program Sum;
{$APPTYPE CONSOLE}
var i, n, summa :integer;

begin
  writeln('vvedite n');
  readln(n);

  summa:=0;

  for i:=1 to n do summa:=summa+i;

  writeln('summa=', summa);
  readln;
end.
```

Прямой цикл выполняется только в том случае, когда начальное значение счетчика цикла меньше или равно конечному значению. Как отмечалось выше, переменная цикла может быть любого порядкового типа, например

Листинг 8

```
program Letters;
{$APPTYPE CONSOLE}
var Letter:char;

begin

  for Letter:='a' to 'z' do write(Letter);

  readln;
end.
```

Программа приведенная в листинге 8 выводит на экран символы от a..z.

### ***Конструкция repeat ... until***

Эту конструкцию называют циклом с постусловием, т.к условие окончания цикла вычисляется после выполнения тела цикла. Конструкция repeat until имеет следующий синтаксис.

**repeat**

<операторы>

**until** <условие>;

Операторы находящиеся между зарезервированными словами **repeat** и **until** повторяются до тех пор пока не выполнится условие (результат вычисления логического выражения не станет равным **true**). Так как условие проверяется в конце конструкции, то тело этого цикла всегда будет выполняться хотя бы один раз. Еще одной особенностью данной конструкции является отсутствие потребности в использовании составного оператора, т. к. зарезервированные слова обрамляют тело цикла.

Перепишем программу приведенную в листинге 7 с использованием цикла **repeat until**.

Листинг 9

```
program Sum2;
```

```
{$APPTYPE CONSOLE}
var i, n, summa :integer;

begin
  writeln('vvedite n');
  readln(n);

  summa:=0;
  i:=1;
  repeat
    summa:=summa+i;
    inc(i);
  until i>=n+1;

  writeln('summa=', summa);
  readln;
end.
```

Количество повторений тела этого цикла не фиксировано и может быть бесконечным, если условие выхода из цикла никогда не выполнится.

### **Конструкция while ...do**

Эта конструкция называется циклом с предусловием, т.к. условие окончания цикла вычисляется перед выполнением тела цикла. Синтаксис этой конструкции следующий

```
while <условие> do <оператор>;
```

Оператор после зарезервированного слова **do** повторяется до тех пор, пока выполняется условие после while. Так как условие окончания цикла проверяется до выполнения тела цикла, то тело цикла может не выполниться ни разу. Количество повторений данного цикла также не фиксировано, и цикл может быть бесконечным.

Перепишем программу из листинга 7 с использованием цикла while do.

Листинг 10

```
program Sum3;

{$APPTYPE CONSOLE}
var i, n, summa :integer;

begin
  writeln('vvedite n');
  readln(n);

  summa:=0;
  i:=1;
  while i>=n do
    begin
      summa:=summa+i;
      inc(i);
    end;

  writeln('summa=', summa);
  readln;
```

end.

## Оператор *break*

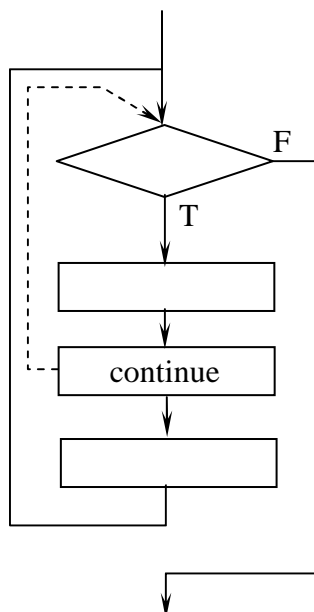
Оператор `break` служит для досрочного завершения цикла. Если этот оператор встречается в теле цикла, то начиная с этого оператора выполнение цикла прерывается, например изменим программу приведенную в листинге 10 следующим образом.

Листинг 11

```
program Sum4;  
  
{$APPTYPE CONSOLE}  
var i, n, summa :integer;  
  
begin  
  writeln('vvedite n');  
  readln(n);  
  
  summa:=0;  
  i:=1;  
  while 1=1 do  
    begin  
      summa:=summa+i;  
      inc(i);  
      if i>=n+1 then break;  
    end;  
  
  writeln('summa=', summa);  
  readln;  
end.
```

Обратите внимание на условие в цикле **while do** оно всегда выполняется, следовательно цикл будет бесконечным. Выход из цикла организуется с помощью функции `break`.

## Оператор *continue*



Оператор `continue` используется для досрочного завершения итерации цикла. Если в теле цикла встречается функция `continue`, то выполнение тела цикла прерывается и начинается следующая итерация цикла.

## Организация циклических конструкций с помощью оператора *goto*

Для примера рассмотрим реализацию цикла `repeat until` с помощью оператора `goto`.

Листинг 12



```

program Sum5;

{$APPTYPE CONSOLE}
label beg_loop, //метка начала цикла
      end_loop; //метка окончания цикла

var i, n, summa :integer;

begin
  writeln('vvedite n');
  readln(n);

  //инициализация переменных
  summa:=0;
  i:=1;

  beg_loop: //начало цикла

    if i>=n+1 then goto end_loop;

    //начало тела цикла
    summa:=summa+i;
    inc(i);
    //конец тела цикла

    goto beg_loop;

  end_loop: //конец цикла

  writeln('summa=', summa);
  readln;
end.

```

Как видно реализация цикла с помощью оператора безусловного перехода **goto** более громоздкая и сложная, чем при помощи конструкции **repeat until**. Но именно в конструкции такого типа компилятор переводит все циклические конструкции так как процессор не содержит команд для организации циклов. Еще более громоздкие конструкции будут при организации вложенных циклов. Для примера рассмотрим организацию функции break. Изменим программу в приведенную в листинге 5 таким образом, чтобы цикл прерывался при достижении переменной summa некоторого значения.

Листинг 13

```

program Sum6;

{$APPTYPE CONSOLE}
label beg_loop, //метка начала цикла
      end_loop; //метка окончания цикла

const max_summa=20;

var i, n, summa :integer;

begin
  writeln('vvedite n');
  readln(n);

  //инициализация переменных
  summa:=0;

```

```

i:=1;
beg_loop: //начало цикла
  if i>=n+1 then goto end_loop;

  //начало тела цикла
  summa:=summa+i;
  //начало оператора break
  if summa>max_summa then goto end_loop;
  //конец оператора break
  inc(i);
  //конец тела цикла

  goto beg_loop;

end_loop: //конец цикла

writeln('summa=', summa);
readln;
end.

```

Программы приведенные в листингах 12 и 13 сложны для анализа. Применение оператора безусловного перехода **goto** в программах на Object Pascal не рекомендуется т.к. все возможные алгоритмы можно организовать с помощью условных и циклических операторов.

Следует отметить, что тело цикла может содержать циклические операторы, поэтому в Object Pascal возможна реализация вложенных циклов. Такие циклы часто используются для работы с многомерными массивами. Пример программы использующей вложенные циклы приведен в листинге 14.

Листинг 14

```

program Sum7;
{$APPTYPE CONSOLE}

var i, j, n, summa :integer;

begin
  writeln('vvedite n');
  readln(n);

  for i:=1 to n do //внешний цикл
  begin
    summa:=0;
    for j:=1 to i do summa:=summa+j; //внутренний цикл
    writeln('summa ', i, '=', summa);
  end;

  readln;
end.

```

## Комментарии в программах

Комментарий это часть текста которая игнорируется компилятором и служит лишь для описания логики работы программы или других целей. Object Pascal поддерживает два вида комментариев однострочные и многострочные.

Однострочные комментарии задаются двумя обратными слешами без пробела (`//`). Все символы введенные после этого символа комментария до конца строки считаются комментарием и не учитываются компилятором, например

```
writeln('vvedite n');
readln(n);

for i:=1 to n do //внешний цикл
begin
  summa:=0;
```

Многострочные комментарии могут располагаться на нескольких строках. Символом начала многострочного комментария является символ '`{`'. Все символы идущие после считаются комментарием и игнорируются компилятором. Конец комментария отмечается символом '`}`', например

```
writeln('vvedite n');
readln(n);

for i{переменная цикла}:=1 to n do
begin
  {это многострочный
  комментарий }
  summa:=0;
```

Обратите внимание, что многострочный комментарий может находиться внутри выражения, но не внутри идентификатора.

## Директивы компилятора

Если многострочный комментарий начинается с символа '`$`', то компилятор рассматривает введенную строку как директиву компилятора

Все директивы условно можно разделить на две группы:

- директивы условной компиляции;
- директивы задающие режимы работы компилятора.

С одной из директив задающих работу компилятора вы уже знакомы – это директива `APPTYPE`, которая задает тип создаваемого приложения консольное (значение `CONSOLE`) или графическое (`GUI`).

Рассмотрим пример применения директив условной компиляции. Допустим что в код программы включен код для отладки программы печатающий промежуточный результат на экран. Естественно в окончательной версии программы этот код не должен попасть в исполняемый модуль. Обычно для реализации этой функции используются следующие три директивы.

```
DEFINE <имя>
```

Объявляет имя. Под именем подразумевается любой идентификатор. Например `{ $DEFINE DEBUG }`

IFDEF <имя>

Проверяет объявлено ли указанное имя. Если оно объявлено, то текст идущий после директивы включается в код программы, иначе он туда не включается.

ENDIF

Указывает на окончание блока IFDEF

Например

Листинг 15

```
program D1rr;
{$APPTYPE CONSOLE}
{$DEFINE DEBUG}
var i, j, n, summa :integer;
begin
writeln('vvedite n');
readln(n);
for i:=1 to n do
begin
summa:=0;
for j:=1 to i do
begin
summa:=summa+j; //внутренний цикл
{$IFDEF DEBUG}
writeln(summa);
{$ENDIF}
end;
writeln('summa ', i, '=', summa);
end;
readln;
end.
```

Для того чтобы отменить объявление имени DEBUG необходимо просто стереть символ “\$” и компилятор пропустит все символы в фигурных скобках, т.к. будет считать его за комментарий. Код находящийся между директивами IFDEF и ENDIF будет пропущен и не попадет в исполняемый модуль. Для включения этого кода в программу достаточно просто восстановить символ ‘\$’.

Рассмотрим еще одну часто используемую директиву компилятора. Это директива INCLUDE. Она имеет следующий синтаксис

INCLUDE <имя файла>

В то место где указана эта директива вставляется все содержимое указанного файла. Допускается сокращенная запись этой директивы {\$I имя файла}

Поместим в файл с именем deb.inc следующую строку writeln(summa); Перепишем предыдущий пример в следующем виде

Листинг 16

```
program D1rr;
{$APPTYPE CONSOLE}
{$DEFINE DEBUG}
```

```

var i, j, n, summa :integer;

begin
writeln('vvedite n');
readln(n);

for i:=1 to n do
begin
summa:=0;
for j:=1 to i do
begin
summa:=summa+j; //внутренний цикл

{$IFDEF DEBUG}
{I deb.inc}
{$ENDIF}
end;
writeln('summa ', i, '=', summa);
end;

readln;
end.

```

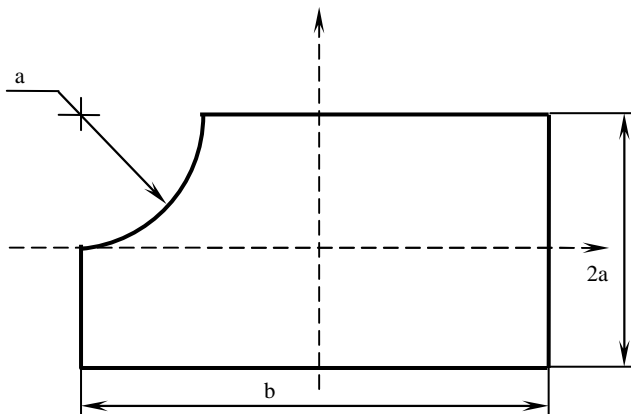
Эта директива применяется для сокращения листинга или включения часто используемого кода.

## Задания к лабораторной работе

1. Наберите все программы приведенные в лабораторной работе.
2. Напишите программу для вычисления значений следующих функций.

$$f(x) = \begin{cases} \cos x, & x \leq \pi \\ 2 \cos x + 5, & x > \pi \end{cases}, f(x) = \begin{cases} 2x + 3, & x < 0 \\ 7x + 1, & 0 \leq x \leq 10 \\ 6x + 2x^2, & x > 10 \end{cases}$$

3. Напишите программу для проверки попадания точки в кольцо. Координаты точки должны задаваться с клавиатуры.
4. Напишите программу для проверки попадания точки в следующую фигуру



5. Напишите программу для печати дней недели по ее номеру. Например, единице соответствует понедельник.

6. Модернизируйте программу, приведенную в листинге 6 таким образом, чтобы если пользователь вводит неверный номер дня недели (т.е. номер меньше единицы и больше 7), то программа запрашивает номер дня недели снова до тех пор, пока не будет введен правильный номер дня недели. Реализуйте эту программу с помощью оператора goto.
7. Решите задачу поставленную в п. 5 с помощью цикла repeat until.
8. Вычислите  $p = \prod_{i=1}^n i$
9. Реализуйте цикл while do с помощью оператора безусловного перехода goto.
10. Реализуйте функцию continue с помощью оператора goto.
11. Напишите программу удаления из строки идущих подряд пробелов. После работы программы в строке между словами должно остаться по одному пробелу. (используйте функции pos и delete, см. предыдущую лабораторную работу).

## Вопросы к лабораторной работе

1. Какие безусловные операторы и функции вам известны?
2. Как работает и для чего нужен составной оператор?
3. Какие условные конструкции вы знаете, чем они отличаются?
4. Какие циклические конструкции вы знаете, в чем их отличие?
5. Каково назначение функции break?
6. Каково назначение функции continue?
7. Как реализуются циклические конструкции на машинном языке?
8. Какие виды комментариев существуют в Object Pascal?
9. Как задать директиву компилятора?
10. Для чего используется условная компиляция, какие директивы компилятора для этого применяются?
11. Для чего применяется директива компилятора INCLUDE?

## Справочные таблицы

Листинг 1.....	16
Листинг 2.....	17
Листинг 3.....	18
Листинг 4.....	18
Листинг 5.....	19
Листинг 6.....	20
Листинг 7.....	22
Листинг 8.....	22
Листинг 9.....	23
Листинг 10.....	23
Листинг 11.....	24
Листинг 12.....	24

Листинг 13.....	25
Листинг 14.....	26
Листинг 15.....	28
Листинг 16.....	28

# Лабораторная работа № 3

## Структурные типы данных. Множества. Массивы.

### Введение

В лабораторной работе рассмотрен тип-множество, основные операторы для работы с этим типом. Рассмотрено описание массивов в языке Object Pascal. Рассмотрены алгоритмы ввода-вывода одно и многомерных массивов, а также алгоритмы вставки и удаления элементов. Рассмотрены некоторые алгоритмы сортировки и поиска данных в массивах.

### Структурные типы данных

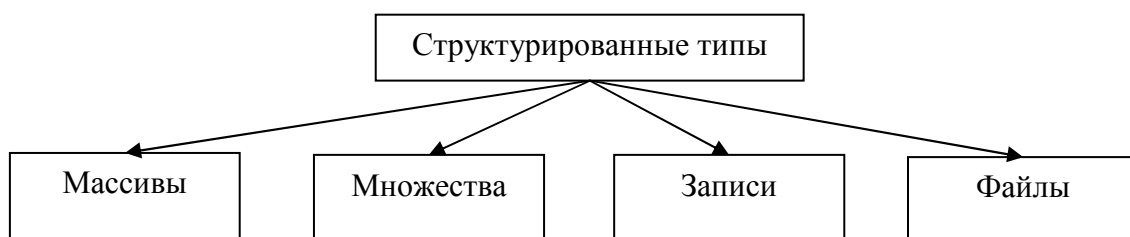


Рисунок 4 – Структурированные типы данных

В Object Pascal определено четыре структурированных типа данных: массивы, записи, множества и файлы (Рисунок 1). Все эти типы характеризуются множественностью образующих их элементов. Каждая переменная структурированного типа содержит в себе несколько компонентов, каждый компонент, в свою очередь, может также принадлежать к структурированному типу. В Object Pascal произвольная глубина вложенности типов, однако размер переменной не может превышать 2 Гб\*.

### Множества

Множества – это наборы однотипных логически связанных друг с другом объектов. Количество элементов, входящих в множество, может изменяться от 0 до 256. Два множества называются эквивалентными если все их элементы одинаковы, причем порядок следования элементов в множестве безразличен. Если все элементы одного множества входят также и в другое, то говорят о включении первого множества во второе. Пустое множество включается в любое другое. Синтаксис описания типа множества имеет вид

`<имя типа> = set of <базовый тип>;`

Здесь имя типа – правильный идентификатор;

set, of – зарезервированные слова (множество, из);

базовый тип – базовый тип элементов множества в качестве которого может использоваться любой порядковый тип число значений которого не превосходит 256. Например:

```
type   TChars      = set of char;
       TDigitsChar = set of '0'..'9';
       TDigits     = set of 0..9;
```

\* Это ограничение связано с ограничениями Windows. Для Windows 3.x это ограничение 64 Кб. Для Windows 9x, Windows NT/2000/XP это ограничение 2Гб.



При объявлении типа TChars в качестве базового типа используется тип char т.к. число различных значений его не превосходит 256. Для остальных двух типов используется тип диапазон. Обратите внимание, что следующее определение типа ошибочно, т.к. тип Integer имеет более 256 различных значений.

```
type TInt=set of Integer;
```

Заполнить множество элементами можно с помощью конструктора множества, конструктор множества имеет следующий синтаксис

```
<ИМЯ МНОЖЕСТВА>:=[<СПИСОК ЭЛЕМЕНТОВ МНОЖЕСТВА>];
```

Например

```
var    s1:TChars;
        s2: TDigits;

begin
...
    s1:=['a'..'d', 'e', 'h'];
    s2:=[1, 2, 5];
...
end.
```

Описание операций определенных над множествами приведено в таблице 1.

Таблица 10 – Операции над множествами

<i>Операция</i>	<i>Описание</i>
*	Пересечение множеств
+	Объединение множеств
-	Разность множеств
=	Проверка эквивалентности, возвращает true если множества эквивалентны
<>	Проверка не эквивалентности
<=	Проверка вхождения, возвращает true, если первое множество включено во второе
>=	Проверка вхождения, возвращает true, если второе множество включено в первое
<b>in</b>	Проверка принадлежности, возвращает true, если элемент принадлежит множеству.

Демонстрация некоторых операций со множествами приведена в листинге 1.

Листинг 17

```
program Sets;
{$APPTYPE CONSOLE}

type TweekDays=(mo, tu, we, th, fr, sa, su);
    TDays=set of TweekDays;

var Days:TDays;
    I :TweekDays;
```

```

begin
Days:=[mo..fr];

//вывод списка рабочих дней
writeln('Rabochie dni:');
for i:=mo to su do
  if i in Days then
    case i of
      mo: writeln('monday');
      tu: writeln('tuesday');
      we: writeln('wednesday');
      th: writeln('thursday');
      fr: writeln('friday');
      sa: writeln('saturday');
      su: writeln('sunday');
    end;

  readln;
end.

```

## Массивы

Отличительной чертой массивов является то, что они содержат компоненты только одного типа (в том числе и структурированного). Эти компоненты легко упорядочить и обеспечить доступ к любому из них указав его порядковый номер, индекс. Описание массива имеет следующий вид

**<имя типа>=array [<список индексов>] of <тип элемента>;**

Здесь <имя типа> – правильный идентификатор;

**array, of** – зарезервированные слова;

<список индексов> – список из одного или нескольких индексных типов;

<тип элемента> – тип элементов массива.

В качестве индексного типа может использоваться любой порядковый тип, кроме типов число значений которых превосходит  $2^{32}$ . Обычно в качестве индексного типа используется тип-диапазон в котором задаются границы изменения индексов. Примеры объявлений массива приведены ниже.

```

type TArray=array [byte] of Integer;
   TAZCharArray=array ['a'..'z'] of char;
   TMyArray=array [1..100] of real;

//массивы можно объявлять сразу в разделе var
var a:array [1..5] of Integer;
    b:array [5..9] of string;

```

Так как тип элемента идущий за of может быть любым, в том числе с структурированным, то он может быть и другим массивом, например

```
TMatrix=array [0..5] of array [-2..7] of real;
```

Такую запись обычно заменяют более компактной

```
TMatrix=array [0..5, -2..7] of real;
```

Глубина вложенности структурированных типов, как отмечалось выше не ограничена, но существует ограничение на максимальный размер переменной в 2 Гб. Массивы

объявленные выше называют многомерными. Трехмерный массив объявляется следующим образом.

```
TMatrix=array [0..5, -2..7, 0..7] of real;
```

В Object Pascal можно одним оператором присваивания передать все элементы одного массива другому массиву того же типа, например.

```
var  
  a,b: array [0..5] of real;  
begin  
  ...  
  a:=b;  
  ...  
end.
```

Следует отметить, что объявление

```
var  
  a: array [0..5] of real;  
  b: array [0..5] of real;
```

создаст разные типы массивов, поэтому при их присваивании возникнет ошибка (несоответствие типов).

Для доступа к элементу массива применяется оператор [], например

```
var  
  a: array [0..5] of real;  
  c: array [0..5, 0..2] of real;  
  
begin  
  a[1]:=1.0;  
  a[2]:=3.5;  
  c[0, 1]:=a[1];  
  
end.
```

Операция сравнения массивов в Object Pascal не определена, сравнить два массива можно поэлементно. Пример программы для сравнения двух массивов приведен в листинге 2.

Листинг 18

```
program Arrays2;  
  
{ $APPTYPE CONSOLE }  
  
const n=10;  
var a, b: array [1..n] of real;  
    equal: Boolean;  
    i: Integer;  
  
begin  
  ...  
  equal:=True;  
  for i:=Low(a) to High(a) do  
    if a[i]<>b[i] then  
      begin  
        equal:=False;  
        break;  
      end;  
  
  if not equal then writeln('Masivi ne ravni');  
  ...
```

**end.**

Обратите внимание на использование функций Low, High при работе с массивами. Для массива функция Low возвращает значение минимального индекса массива, а функция High значение максимального индекса массива.

## Ввод-вывод массивов

Программа приведенная в следующем листинге осуществляет ввод и вывод данных из массива.

Листинг 19

```
program InOut;
{$APPTYPE CONSOLE}
const n=5; //длина массива
var a:array [1..n] of real;
    i:Integer;

begin
    //Ввод одномерного массива
    for i:=Low(a) to High(a) do
        begin
            write('a[' , i , ']=');
            readln(a[i]);
        end;
    {здесь происходит обработка данных массива}
    //Вывод элементов одномерного массива на экран
    for i:=Low(a) to High(a) do
        writeln('a[' , i , ']=', a[i]:6:2);

    readln;
end.
```

Для ввода двумерных массивов можно использовать следующий код

Листинг 20

```
program InOut2;
{$APPTYPE CONSOLE}
const n=5; //число строк матрицы
      m=2; //число столбцов
var a:array [1..n, 1..m] of real;
    i, j:Integer;

begin
    //Ввод массива
    for i:=1 to n do
        for j:=1 to m do
            begin
                write('a[' , i , ', ' , j , ']=');
                readln(a[i, j]);
            end;{for j}
        end;
    end;
```

```

{Здесь происходит обработка данных массива}

//Вывод элементов массива на экран
for i:=1 to n do
begin
  for j:=1 to m do
    write(' a[' , i , ', ' , j , ']=', a[i, j]:6:2);

    writeln;
  end;{for i}

readln;
end.

```

### Вставка и удаление элемента в массив

Для начала рассмотрим вставку элемента в массив. Для простоты будем рассматривать только одномерные массивы. Массив состоит из некоторого количества элементов  $n$  (емкость массива). Текущее количество элементов массива находится в переменной  $n$ .

Перед вставкой очередного элемента проверяем, что текущее количество элементов массива меньше, чем его емкость.

Далее проверяем, вставляется ли элемент в конец массива или нет. Если элемент вставляется в конец массива, то увеличиваем  $n$  на единицу и добавляем элемент. Иначе сдвигаем элементы массива индекс которых больше или равен индексу вставляемого элемента.

Приведенный алгоритм реализован в программе приведенной в листинге 5.

```

program InSEIt;
{$APPTYPE CONSOLE}

const nmax=5; //емкость массива

var a:array [1..nmax] of real; //массив
    element:real; //элемент для вставки
    index :integer; //индекс элемента для вставки
    n :integer; //число элементов массива
    i :integer;

begin
  writeln('vvedite chislo elementov massiva');
  readln(n);

  //Ввод массива
  for i:=1 to n do
  begin
    write('a[' , i , ']=');
    readln(a[i]);
  end;

  //Элемент для вставки
  writeln('vvedite element');
  readln(element);
  //Индекс элемента
  writeln('vvedite index elementa');
  readln(index);

```

```

//Вставка элемента
if index<n+1 then
begin
inc(n); //увеличиваем длину массива
if (Low(a)<=index)and(index<=n) then
for i:=n downto index do a[i]:=a[i-1]; //сдвиг массива

a[index]:=element;
end
else
begin
writeln('Invalid index');
readln;
halt(0);
end;

//Вывод элементов массива на экран
for i:=1 to n do writeln('a[', i, ']=' , a[i]:6:2);

readln;
end.

```

Удаление элемента происходит аналогично. Сначала проверяется, что индекс элемента не выходит за диапазон допустимых значений, а затем сдвигаем элементы таким образом, чтобы закрыть удаляемый элемент.

```

program DelElt;

{$APPTYPE CONSOLE}

const nmax=5; //емкость массива

var a:array [1..nmax] of real; //массив
    index:integer; //индекс элемента для вставки
    n :integer; //число элементов массива
    i :integer;

begin
writeln('vvedite chislo elementov massiva');
readln(n);

//Ввод массива
for i:=1 to n do
begin
write('a[', i, ']=' );
readln(a[i]);
end;

//Индекс элемента
writeln('vvedite index elementa');
readln(index);

//Удаление элементов
if index<n+1 then
begin
if (Low(a)<=index)and(index<=n) then
for i:=index to n do a[i]:=a[i+1]; //сдвиг массива
dec(n); //уменьшаем длину массива
end
else

```

```

begin
  writeln('Invalid index');
  readln;
  halt(0);
end;

//Вывод элементов массива на экран
for i:=1 to n do writeln('a[' , i, ']=' , a[i]:6:2);

readln;
end.

```

## Поиск

Поиск – это действие, заключающееся в просмотре набора элементов и выделении из этого набора интересующего элемента. Если элементы в массиве не отсортированы, то может использоваться только один алгоритм поиска – последовательный поиск. В случае если массив отсортирован, то возможен бинарный поиск.

### Последовательный поиск.

Последовательный поиск заключается в выборе каждого элемента массива и сравнения его с искомым. Быстродействие алгоритма пропорционально числу элементов. Реализация этого алгоритма с помощью цикла **for ... to ... do** приведена в листинге 5

Листинг 21

```

program Search;

{$APPTYPE CONSOLE}

const nmax=5; //емкость массива

var a:array [0..nmax] of real; //массив
    index:integer; //индекс искомого элемента
    value:real; //значение элемента для поиска
    i :integer;

begin
  writeln('Vvedite massiv ', nmax+1, 'elements');

  //Ввод массива
  for i:=1 to nmax do
    begin
      write('a[' , i, ']=' );
      readln(a[i]);
    end;

  //элемент для поиска
  writeln('Input element for search');
  readln(value);

  //поиск
  index:=-1;
  for i:=Low(a) to High(a) do
    if a[i]=value then
      begin
        index:=i;
        break;
      end;
  end;

```

```

if index>=0 then //если элемент найден
  writeln('Element found in ', index, ' position')
else //если элемент не найден
  writeln('Element not found!');

readln;
end.

```

## Бинарный поиск

Алгоритм бинарного поиска применим только для отсортированного массива. Бинарный поиск работает следующим образом. Берем средний элемент массива и проверяем равен ли он искомому элементу, если равен, то поиск окончен. В противном случае, если искомый элемент меньше среднего, то можно сказать, что элемент присутствует в массиве, то он находится в первой его половине. С другой стороны, если искомый элемент больше среднего, то он находится во второй части массива. Таким образом, одним сравнением удалось локализовать искомый элемент в массиве вдвое меньшей размерности. Описанная операция повторяется до тех пор, пока искомый элемент не будет найден (если он присутствует в массиве). Так как при каждом выполнении цикла размер массива уменьшается в два раза, то быстродействие алгоритма будет пропорционально  $\log_2 n$ . Так, например для поиска в массиве из 256 элементов понадобится произвести всего 8 сравнений. Реализация алгоритма приведена в листинге 6.

Листинг 22

```

program Search2;

{$APPTYPE CONSOLE}
const n=10; //длина массива
var a:array [0..n-1] of integer=(1,2,3,4,5,16,17,56,78,110);
    l, //левый индекс
    r, //правый индекс
    m:integer; //индекс среднего элемента
    e1:integer; //значение искомого элемента
begin
  write('vvedite element dlia poiska ');
  readln(e1);
  l:=0;
  r:=n-1;

  while l<=r do
    begin
      m:=(l+r) div 2; //вычисляем индекс среднего элемента

      {если значение среднего элемента меньше искомого значения,
      то переместить левый индекс на позицию до среднего индекса}
      if a[m]<e1 then l:=succ(m)
      {если значение среднего элемента больше искомого значения,
      то переместить правый индекс на позицию после среднего
      индекса}
      else
        if a[m]>e1 then r:=pred(m)
        else //иначе элемент найден
          begin
            writeln('Search result a[' , m, ']=' , a[m]);
            break;
          end
    end
end.

```



```

        end;
    end; {while}

    if l>=r then writeln('Element ', e1, ' not found');
    readln;
end.

```

## Поиск максимального и минимального элементов массива

Поиск минимального или максимального элемента массива можно осуществить с помощью алгоритма приведенного в листинге 7.

Листинг 23

```

program MinMax;

{$APPTYPE CONSOLE}
const n=10; //длина массива
var a:array [0..n-1] of integer;
    i:integer;
    min, //значение минимального элемента
    max:integer; //значение максимального элемента
begin

    //инициализируем массив случайными числами
    Randomize;
    for i:=0 to n-1 do a[i]:=random(10);

    max:=a[0];
    min:=a[0];
    for i:=1 to n-1 do
        begin
            if a[i]>max then max:=a[i];
            if a[i]<min then min:=a[i];
        end; {for}

    writeln('min ', min);
    writeln('max ', max);
    readln;
end.

```

## Сортировка

Сортировка это упорядочение данных в соответствии с заданными критериями. С данными удобнее работать если они отсортированы. Например алгоритм бинарного поиска работает гораздо быстрее алгоритма последовательного поиска. Существуют десятки различных видов сортировки. Каждый со своими достоинствами и недостатками. Рассмотрим некоторые простейшие алгоритмы сортировки.

### Пузырьковая сортировка

Это первый вид сортировки с которым сталкиваются программисты при изучении программирования. Это самый медленный вид сортировки, но и самый простой для реализации алгоритм сортировки.

В качестве примера рассмотрим сортировку массива по возрастанию. Работает алгоритм следующим образом, проверяем два соседних элемента если первый элемент больше второго, то переставляем их местами. Продолжаем выполнять выше описанные

действия до тех пор пока не будет отсортирован весь массив. В наихудшем случае минимальный элемент будет находиться в конце массива. Так как, за одну итерацию элемент перемещается только на одну позицию, то потребуется  $n-1$  итерация для сортировки всего массива. Программа реализующая данный алгоритм приведена в листинге 8.

Листинг 24

```
program BubbleSort;
{$APPTYPE CONSOLE}
const n=10; //длина массива
var a:array [0..n-1] of integer;
    i, j:integer;
    buf:integer;
begin
    //инициализируем массив случайными числами
    Randomize;
    for i:=0 to n-1 do a[i]:=random(100);

    //выводим значения не отсортированного массива
    writeln('Source array');
    for i:=0 to n-1 do write(a[i], ' ');
    writeln;
    writeln;

    //сортируем массив
    for i:=0 to n-1 do
        for j:=0 to n-2 do
            if a[j]>a[j+1] then
                begin
                    buf:=a[j];
                    a[j]:=a[j+1];
                    a[j+1]:=buf;
                end;

    writeln('Sort array');
    //выводим значения отсортированного массива
    for i:=0 to n-1 do write(a[i], ' ');

    readln;
end.
```

Приведенный ниже алгоритм можно немного улучшить следующим образом. Если массив будет отсортирован раньше, чем за  $n-1$  итерацию, то можно досрочно завершить внешний цикл. Приведенный алгоритм реализован в программе приведенной в листинге 9.

Листинг 25

```
program BubbleSort2;
{$APPTYPE CONSOLE}
const n=10; //длина массива
var a:array [0..n-1] of integer;
    i:integer;
    flag:Boolean;
    buf:integer;
begin
```

```

//инициализируем массив случайными числами
Randomize;
for i:=0 to n-1 do a[i]:=random(100);

//выводим значения не отсортированного массива
writeln('Source array');
for i:=0 to n-1 do write(a[i], ' ');
writeln;
writeln;

//сортируем массив
flag:=False;
while not flag do
begin
flag:=True;
for i:=0 to n-2 do
if a[i]>a[i+1] then
begin
buf:=a[i];
a[i]:=a[i+1];
a[i+1]:=buf;
flag:=False;
end;{for}
end;{while}

writeln('Sort array');
//выводим значения отсортированного массива
for i:=0 to n-1 do write(a[i], ' ');

readln;
end.

```

## Сортировка методом выбора

Начиная с первого элемента массива ищем минимальный элемент массива и меняем его местами с первым элементом. Далее начиная со второго элемента ищем минимальный элемент массива и меняем его местами со вторым элементом. Процесс продолжается до тех пор, пока весь массив не будет отсортирован. Приведенный алгоритм реализован в программе в листинге 26

Листинг 26

```

program SelectSort;

{$APPTYPE CONSOLE}
const n=10; //длина массива
var a:array [0..n-1] of integer;
    i, j:integer;
    buf:integer;
    minIndex:Integer;
begin

//инициализируем массив случайными числами
Randomize;
for i:=0 to n-1 do a[i]:=random(100);

//выводим значения не отсортированного массива

```

```

writeln('Source array');
for i:=0 to n-1 do write(a[i], ' ');
writeln;
writeln;

//сортируем массив
for i:=0 to n-2 do
begin
  minIndex:=i;
  for j:=i+1 to n-1 do
    if a[j]<a[minIndex] then minIndex:=j;

  if minIndex<>i then
  begin
    buf:=a[i];
    a[i]:=a[minIndex];
    a[minIndex]:=buf;
  end;
end;{for i}

writeln('Sort array');
//выводим значения отсортированного массива
for i:=0 to n-1 do write(a[i], ' ');

readln;
end.

```

Рассмотрение других методов сортировок продолжим в последующих лабораторных работах.

## Задания к лабораторной работе

1. Наберите и отладьте программы приведенные в лабораторной работе.
2. Напишите программу для нахождения среднего значения массива по формуле

$$a_{cp} = \frac{\sum_{i=0}^n a_i}{n}$$

3. Напишите программу для вычисления скалярного произведения векторов.
4. Напишите программу для умножения двух матриц.
5. Реализуйте алгоритм последовательного поиска с помощью цикла while do.
6. Реализуйте алгоритм, приведенный в листинге 9, только с помощью цикла for to do.
7. Напишите программу, реализующую ввод и вывод списка студентов, а также его сортировку.

## Вопросы к лабораторной работе

1. Как задается тип множество?
2. Какие операторы для работы с множествами вы знаете?
3. Как задать массив в Object Pascal?
4. Объясните алгоритм работы бинарного поиска?
5. Объясните алгоритм пузырьковой сортировки?
6. Объясните алгоритм сортировки методом выбора?

7. Сколько перестановок в худшем случае надо сделать при применении алгоритма сортировки методом выбора?
8. Сколько сравнений в худшем случае нужно сделать при применении алгоритма пузырьковой сортировки?

## **Справочные таблицы**

Таблица 1 – Операции над множествами.....	33
---	----

# Лабораторная работа №4

## Структурные типы данных. Записи.

### Организация таблиц в памяти.

## Введение

В лабораторной работе рассматривается работа с типом запись. Рассматриваются также некоторые методы работы с данным типом.

## Тип запись

Запись – это структура данных, состоящая из фиксированного числа компонентов, называемых полями записи. В отличие от других структурных типов, тип запись может содержать в себе данные различных типов. Синтаксис типа запись имеет следующий вид

<идентификатор типа>=**record**

<имя поля>:<тип поля>;

<имя поля>:<тип поля>;

...

**end;**

где <идентификатор типа> – правильный идентификатор;

**record** – зарезервированное слово;

<имя поля> – правильный идентификатор, уникальный в пределах объявления типа запись;

<тип поля> – любой тип **Object Pascal**, в том числе и структурный.

Рассмотрим пример объявления типа запись для описания информации о некоторой персоне.

```
type TPerson=record
    name   :string[12]; //имя
    familia:string[16]; //фамилия
    Age:byte;           //возраст
end;
```

Значения переменных типа запись можно присваивать переменным того же типа.

```
var a, b:TPerson;

begin
    ..
    a:=b;
    ..
end.
```

Для того чтобы получить доступ к каждому из компонентов записи необходимо использовать составное имя. Для этого после имени переменной необходимо поставить точку и имя поля.

```
var a, b:TPerson;

begin
    ..
    a.name:='Иван';
    a.familia:='Иванов';
    a.age:=18;
```

```
end.
```

Так как поле в типе запись может быть любого типа, в том числе и быть записью, то для таких полей необходимо продолжать уточнения, например

```
type TBirthDay=record
    Day,           //день
    Month:byte;   //месяц
    Year :word;   //год
end;

TPerson=record
    name :string[12]; //имя
    familia:string[16]; //фамилия
    BirthDay:TBirthDay; //дата рождения
end;

var Person:TPerson;

begin
...
    Person.name:='Иван';
    Person.familia:='Иванов';
    Person.BirthDay.Day:=1;
    Person.BirthDay.Month:=1;
    Person.BirthDay.Year:=2000;
...
end.
```

Для упрощения доступа к полям записи можно использовать следующий оператор

**with** <переменная> **do** <оператор>;

где **with**, **do** – зарезервированные слова;

<переменная> – переменная типа запись, за которой, возможно следует список вложенных полей;

<оператор> – любой оператор **Object Pascal**.

Для примера перепишем фрагмент программы приведенный выше с использованием оператора **with do**.

```
var Person:TPerson;

begin
...
with Person do
begin
    name:='Иван';
    familia:='Иванов';
    BirthDay.Day:=1;
    BirthDay.Month:=1;
    BirthDay.Year:=2000;
end;
...
end.
```

В выше приведенном фрагменте можно применить еще раз оператор **with do**.

```
var Person:TPerson;

begin
...
```

```

with Person do
begin
  name:='Иван';
  familia:'Иванов';

  with Birthday do
  begin
    Day:=1;
    Month:=1;
    Year:=2000;
  end;

end;
...
end.

```

**Object Pascal** разрешает использовать в записи так называемые вариантные поля. Синтаксис определения вариантных полей приведен ниже.

```

case <порядковый тип> of
<значение>: (
  <имя поля>:<тип поля>;
  <имя поля>:<тип поля>;
...);
< значение >: (
  <имя поля>:<тип поля>;
  <имя поля>:<тип поля>;
...);
...

```

Замечание: Вариантная часть записи должна всегда находиться в конце записи после всех фиксированных полей. Запись может иметь только одну вариантную часть. Вариантная секция в записи является не обязательной и обычно отсутствует.

Пример использования записей с вариантной частью приведен ниже.

```

type TPerson=record
  name :string[12]; //имя
  familia:string[16]; //фамилия
  case byte of//род занятий
    0:(Type_computer:string[20]; //компьютеры
      MByte:Integer;
      Compatible:Boolean);
    1:Instrument: array[1..3] of string[10]; //музыка
end;

```

В данном примере определено два варианта: первый вариант предназначен для человека увлекающегося компьютерами, а второй для человека увлекающегося музыкой. Особенностью вариантных полей является то, что нельзя использовать одновременно поля для различных вариантов, т.к. им выделяется одна и та же область памяти. Вариантные поля обычно используют для хранения взаимоисключающей информации с целью экономии памяти. На месте ключа в конструкции case of может стоять любой встроенный или пользовательский порядковый тип. Выбор этого типа не имеет значения и по сути игнорируется компилятором.

Обращение к вариантным полям происходит также как и к фиксированным, например



```

var Person:TPerson;

begin
  ...
  with Person do
    begin
      name:='Иван';
      familia:='Иванов';
      Type_computer:='AMD Athlon';
      MByte:=1024;
      Compatible:=true;
    end;
  ...
end.

```

### **Ввод и вывод переменных типа запись**

Для операций ввода-вывода для записей в Object Pascal не предусмотрено ни каких средств, поэтому ввод и вывод данных для записей необходимо реализовывать самостоятельно. Ниже приведен пример организации ввода и вывода записи.

Листинг 27

```

program InOutRec;

{$APPTYPE CONSOLE}

type TStudent=record
  name   :string[12]; //имя
  familia:string[16]; //фамилия
  grupper:string[4]; //группа
end;

var Student:TStudent;
begin
  writeln('vvedite dannie');

  //ВВОД ЗАПИСИ
  with Student do
    begin
      write('name '); readln(name);
      write('familia '); readln(familia);
      write('grupper '); readln(grupper);
    end;

  //ВЫВОД ЗАПИСИ
  with Student do
    begin
      writeln(name);
      writeln(familia);
      writeln(grupper);
    end;

  readln;
end.

```

Как видно из примера каждое поле записи необходимо вводить (выводить) отдельно.

## Организация таблиц

Так как запись может содержать данные различных типов, то ее удобно использовать для хранения табличных данных. Действительно, можно заметить, что переменная типа запись представляет собой строку таблицы. Соответственно массив записей будет представлять собой таблицу. Ниже рассмотрен простой пример программы для работы с таблицей.

Листинг 28

```
program TableRec;
{$APPTYPE CONSOLE}
const n=10; //емкость массива
type TStudent=record
    name :string[12]; //имя
    familia:string[16]; //фамилия
    grupper :Integer; //группа
end;

var Table:array [1..n] of TStudent; //таблица
    Count:integer; //число строк в таблице
    MenuState:byte; //код выбранного действия
    i:integer;
begin
    Count:=0;

    repeat
        //Выводим меню
        writeln;
        writeln('viberite deistvie');
        writeln('0 - Exit program');
        writeln('1 - Add record');
        writeln('2 - Show all record');
        write('>>'); readln(MenuState);

        //выбираем действие
        case MenuState of
            1:
                begin //добавление записи в конец массива
                    if Count=n then //если число элементов превышает емкость
                    массива
                        writeln('Error: array overflow')
                    else //если в массиве есть место, то добавляем запись
                        begin
                            inc(Count);

                            writeln('Insert record >>');
                            with Table[Count] do
                                begin
                                    write('name :'); readln(name);
                                    write('familia :'); readln(familia);
                                    write('grupper :'); readln(grupper);
                                end; {with}
                            end; {else}
                        end; {add}

            2:
                begin //вывод всех записей на экран
                    //рисуем таблицу
```

```

writeln('|-----|');
for i:=1 to Count do
  with Table[i] do
    writeln(i, ' ', name, ' ', familia, ' ', группа);
  writeln('|-----|');
end;{show}
end;{case}
until MenuState=0;

writeln('Press Enter to exit');
readln;
end.

```

## Задания к лабораторной работе

1. Наберите и отладьте программы приведенные в лабораторной работе.
2. Измените программу приведенную в листинге 2. Добавьте следующие функции в программу:
  - удаление произвольной записи из таблицы;
  - сортировка таблицы по столбцу «фамилия»;
  - вывести только студентов относящихся к заданной группе (номер группы вводится).

## Вопросы к лабораторной работе

1. Какое зарезервированное слово применяется для описания типа запись?
2. Как получить доступ к полям записи?
3. Как создать вариантную часть записи?
4. Почему нельзя одновременно использовать поля записи относящиеся к различным вариантам в вариантной части записи, к чему может привести неправильное использование вариантных полей?
5. Каким образом осуществляется ввод-вывод переменных типа запись?
6. Какую функцию необходимо использовать для вычисления размера записи в байтах?

## Справочные таблицы

## Лабораторная работа №5

### Структурные типы данных. Файлы.

#### Введение

В данной лабораторной работе рассмотрены основные типы файлов Object Pascal: текстовые файлы, типизированные файлы и нетипизированные файлы. Рассмотрены основные приемы обработки ошибок ввода-вывода.

#### Файлы

В **Object Pascal** реализована достаточно высоко уровневая поддержка файлов. **Object Pascal** поддерживает наиболее часто применяемые разновидности файлов. Всего в **Object Pascal** введено три вида файлов:

- текстовые файлы;
- типизированные файлы;
- нетипизированные файлы.

Работа с файлами в **Object Pascal** едина для трех основных типов файлов и очень простая. Ведется она через файловую переменную, одного из трех типов, к которой применяются функции и процедуры. Типовая последовательность следующая:

- Объявляется файловая переменная нужного типа;
- С этой файловой переменной связывается файл, функцией AssignFile;
- Затем файл открывается Reset/Rewrite/Append;
- Производятся операции чтения или записи, разновидности Read/Write;
- Файл закрывается с помощью функции CloseFile.

Основные функции для работы с файлами приведены в таблице 1.

Таблица 11 – Основные функции для работы с файлами

<i>Наименование</i>	<i>Описание</i>
<b>procedure</b> AssignFile(var F; FileName:string);	Связывает файловую переменную F с именем файла FileName
<b>procedure</b> CloseFile(var F);	Закрывает файл.
<b>function</b> IOResult:Integer	Возвращает условный признак последней операции ввода-вывода.
<b>procedure</b> Reset (var F; [; RecSize:Word])	Открывает существующий файл. RecSize имеет смысл только для нетипизированных файлов и указывает размер блока.
<b>procedure</b> Rewrite (var F; [; RecSize:Word])	Создает новый файл. Если файл существовал, то он перезаписывается. RecSize имеет смысл только для нетипизированных файлов и указывает размер блока.

**function** EOF(var F):Boolean;

Возвращает True если достигнут конец файла и False в противном случае.

## Обработка ошибок ввода-вывода

В **Object Pascal** существует два способа обработок ввода-вывода. Первый способ обработки существует еще со времен **Turbo Pascal** и будет рассматриваться ниже. Второй способ основан на механизме обработки исключительных ситуаций и появился в **Object Pascal**. Этот способ считается более прогрессивным и будет рассмотрен позднее.

**Object Pascal** автоматически вставляет в программу код контроля ошибок. При возникновении любой ошибки при работе с функциями ввода-вывода происходит вывод сообщения об ошибке и аварийное завершение программы. Для управления ошибками ввода-вывода сначала необходимо отключить автоматический контроль ошибок. Это делается с помощью директивы компилятора **{\$I-}**. Включить автоматический контроль ошибок можно с помощью директивы компилятора **{\$I+}**. После выполнения очередной функции ввода-вывода необходимо вызвать функцию **IOResult** которая возвращает код ошибки при выполнении последней функции ввода-вывода. Если ошибки не было, то функция возвращает нулевой результат. Некоторые коды ошибок ввода-вывода приведены в таблице 2.

Таблица 12 – Коды ошибок ввода-вывода

<i>Код ошибки</i>	<i>Описание ошибки</i>
100	Ошибка чтения с диска
101	Ошибка записи на диск
102	Имя файла не сопоставлено файловой переменной
103	Файл не открыт
104	Файл не открыт для чтения
105	Файл не открыт для записи
106	Ошибка конвертации при чтении из файла

Ниже приведен пример программы реализующей обработку ошибок ввода-вывода при открытии файла или создании файла.

```
program InOutTemplate;  
{ $APPTYPE CONSOLE }  
var F:TextFile;  
    error:integer;  
begin  
    { сопоставляем имя файла файловой переменной }  
    AssignFile(F, 'test.txt');  
    { $I- } //отключает контроль ошибок ввода-вывода  
    Reset(F); //открываем файл, здесь может произойти ошибка  
  
    { сохраняем значение возвращенное функцией  
    IOResult в переменной error для последующего анализа ошибки }  
    error:=IOResult;  
    //проверяем код последней ошибки  
    if error<>0 then //если произошла ошибка
```

```

begin
  {здесь может анализироваться причина ошибки
  и выводится сообщение об ошибке}
  halt(1); //досрочное завершение программы
end
else //если ошибки не было
begin
  {$I+} //включение контроля ошибок ввода-вывода

  {здесь осуществляется работа с файлом чтение/запись}

end;

closeFile(F); //закрываем файл
end.

```

## Текстовые файлы

Текстовые файлы являются подмножеством двоичных файлов, но в отличие от двоичных не могут содержать весь набор символов. Вся информация в файле разбивается на строки, ограниченные символами возврата каретки (CR) и перевода строки (LF). Допустимые символы это символы с кодами от 32 до 255, символы с кодами ниже 32 являются управляющими и допустимы только следующие коды:

Таблица 13 – Управляющие символы

<i>код символа</i>	<i>наименование символа</i>	<i>производимое действие</i>
08	BS	возврат на шаг
09	TAB	табуляция
0A	LF	перевод строки
0C	FF	перевод листа
0D	CR	возврат каретки
1A	EOF	конец файла

**Object Pascal** поддерживает работу с такими файлами, через файловую переменную типа `TextFile`, где основной единицей является строка, состоящая из основных базовых типов (в текстовом виде, разделенных пробелом), наиболее часто это просто строка, как набор символов. Основные функции применяемые для работы с текстовыми файлами приведены в таблице 4.

Таблица 14 – Подпрограммы для работы с текстовыми файлами

<i>Наименование</i>	<i>Описание</i>
<b>function</b> Eoln (var F:TextFile):boolean;	Возвращает True если достигнут конец строки.
<b>function</b> read (var F:TextFile; V1, V2, ...);	Читает из текстового файла последовательность переменных V1, V2, ... типа Char, String, а также любого целого или вещественного типа. Признаки конца строки игнорируются.

<b>function</b> readln ( <b>var</b> F:TextFile; V1, V2, ...);	Читает из текстового файла последовательность переменных V1, V2, ... типа Char, String, а также любого целого или вещественного типа. Учитываются границы строк
<b>function</b> SeekEof ( <b>var</b> F:TextFile):boolean;	Пропускает все пробелы, знаки табуляции и маркеры конца строки до маркера конца файла или до первого значащего символа. Возвращает True если обнаружен конец файла.
<b>function</b> SeekEoln ( <b>var</b> F:TextFile):boolean;	Пропускает все пробелы, знаки табуляции до маркера конца строки или до первого значащего символа. Возвращает True если обнаружен конец строки.
<b>function</b> write ( <b>var</b> F:TextFile; V1, V2, ...);	Записывает переменные V1, V2, ... в текстовый файл
<b>function</b> writeln ( <b>var</b> F:TextFile; V1, V2, ...);	Записывает переменные V1, V2, ... и признак конца строки в текстовый файл.
<b>procedure</b> Append( <b>var</b> F: Text);	Открывает существующий текстовый файл для добавления данных. Добавление данных происходит в конец.

Для примера рассмотрим программу которая осуществляет запись данных нескольких различных типов в текстовый файл.

Листинг 29

```

program TextInOut;
{$APPTYPE CONSOLE}
var F:TextFile; //файловая переменная
    i:integer;
    r:real;
    error:Integer;
begin
    i:=2;
    r:=10.56;

    {$I-}
    AssignFile(F, 'test.txt');
    Rewrite(F);

    error:=IOResult; //проверяем код последней ошибки
    if error<>0 then //если произошла ошибка
        begin
            halt(1); //досрочное завершение программы
        end
    else //если ошибки не было
        begin
            {$I+} //включение контроля ошибок ввода-вывода

```

```

writeln(F, 'Текстовый файл');
writeln(F, i);
writeln(F, r:6:2);
end;

CloseFile(F);
end.

```

Откройте созданный файл test.txt в любом текстовом редакторе и убедитесь что данные в него записаны.

Программа приведенная на листинге 2 осуществляет вывод содержимого текстового файла на экран.

Листинг 30

```

program TextOut;

{$APPTYPE CONSOLE}
var F:TextFile; //файловая переменная
    error:Integer;
    FileName, S:String;
begin
  writeln('vvedite imia faila');
  readln(FileName);

  {$I-}
  AssignFile(F, FileName);
  Reset(F);

  error:=IOResult; //проверяем код последней ошибки
  if error<>0 then //если произошла ошибка
    begin
      writeln('File not exist');
      writeln('Press Enter to exit');
      readln;
      halt(1); //досрочное завершение программы
    end
  else //если ошибки не было
    begin
      {$I+} //включение контроля ошибок ввода-вывода
      while not EOF(F) do
        begin
          readln(F, S);
          writeln(S);
        end;
      end;

  CloseFile(F);
  readln;
end.

```

## Типизированные файлы

Второй тип файлов Object Pascal – это типизированные файлы. Это такой вид файлов, в котором содержатся записи одного типа и фиксированной длины. Часто используется или для организации мини баз, конфигураций, иногда для импорта/экспорта в специальных форматах. Работа с такими файлами не сложнее, чем работа с текстовыми файлами, наряду с освоенными методами добавляется только одно новое свойство. Если



текстовые файлы чисто последовательные, то в типизированных файлах можно перемещаться на любую запись и затем производить последовательное чтение или запись. По сути типизированный файл аналогичен одномерному массиву, только этот массив располагается не в оперативной памяти, а на диске. Типизированный файл определяется следующим образом

**var**

**FileVar: file of <тип>;**

Здесь тип это или предопределенный или пользовательский типы. В качестве типов не могут фигурировать динамические структуры, такие как динамические массивы, длинные строки или любые указатели, поскольку все записи должны быть одинаковой длины и не должны указывать на внешние данные. Для обработки таких данных надо использовать не типизированные файлы. Примеры объявления переменных типа типизированный файл приведены ниже.

**type TPerson=record**

    name:string[16];

    fam:string[24];

    age:Integer;

**end;**

**var F:file of integer;**

**F1:file of real;**

**F2:file of TPerson;**

Для работы с типизированными файлами предназначены подпрограммы приведенные в таблице 5.

**Таблица 15 – Подпрограммы для работы с типизированными файлами**

<i>Наименование</i>	<i>Описание</i>
<b>function</b> read ( <b>var</b> F; V1, V2, ...);	Читает из типизированного файла F последовательность переменных V1, V2, ... того же типа, что и компоненты файла.
<b>function</b> write ( <b>var</b> F; V1, V2, ...);	Записывает переменные V1, V2, ... в типизированный файл.
<b>procedure</b> Seek ( <b>var</b> F; N: Longint);	Смещает указатель файла F к требуемому компоненту: N – номер компонента файла.
<b>function</b> FileSize ( <b>var</b> F):LongInt;	Возвращает количество компонентов файла.
<b>function</b> FilePos ( <b>var</b> F):LongInt;	Возвращает текущую позицию в файле.
<b>procedure</b> Truncate( <b>var</b> F);	Удаляет все компоненты из файла.

**Примечание:** Обратите внимание на следующие особенности типизированных файлов:

- записи считаются с нуля;
- функция **FileSize** возвращает именно количество записей, а не длину файла, как это следует из ее названия;
- **Rewrite** - создает новый файл для чтения/записи, если такой файл существует, его длина устанавливается в ноль
- **Reset** – открывает файл для чтения/записи и не изменяет его длины.

В качестве примера рассмотрим программу для ввода данных в одномерный массив располагающийся на диске.

Листинг 31

```
program SimpleRec;
{$APPTYPE CONSOLE}
const FileName='RecFile.dat';

var F:File of Integer; //файл состоящий из целых чисел
    Buf, i:Integer;
    Count:Integer; //число элементов

begin
  AssignFile(F, FileName);;
  {$I-}
  Rewrite(F);

  //если ошибка при создании файла, то выход из программы
  if IOResult<>0 then Halt(1);
  {$I+}

  writeln('Vvedite chislo elementov');
  readln(Count);

  for i:=0 to Count-1 do
    begin
      write('element[' , i , ']=');
      readln(buf);
      write(F, Buf);
    end;

  CloseFile(F);
end.
```

В следующем листинге приведена простая программа осуществляющая вывод содержимого файла созданного программой в листинге 3 на экран.

Листинг 32

```
program SimpleRec2;
{$APPTYPE CONSOLE}
const FileName='RecFile.dat';

var F:File of Integer; //файл состоящий из целых чисел
    Buf, i:Integer;
    Count:Integer; //число элементов
```

```

begin
AssignFile(F, FileName);;
{$I-}
Reset(F);

//если ошибка при открытии файла, то выход из программы
if IOResult<>0 then Halt(1);
{$I+}

//определяем число записей в файле
Count:=FileSize(F);

for i:=0 to Count-1 do
begin
read(F, Buf);
writeln('element[' , i, ']=' , Buf);
end;

CloseFile(F);
readln;
end.

```

В следующей программе демонстрируется применение подпрограммы seek. Программа выводит значение заданного элемента массива на экран.

### Листинг 33

```

program SimpleSeek;

{$APPTYPE CONSOLE}
const FileName='RecFile.dat';

var F:File of Integer; //файл состоящий из целых чисел
    Buf, i:Integer;
    Count:integer; //число элементов
    index:Integer;
begin
AssignFile(F, FileName);;
{$I-}
Reset(F);

//если ошибка при открытии файла, то выход из программы
if IOResult<>0 then Halt(1);
{$I+}

//определяем число записей в файле
Count:=FileSize(F);

write('vvedite nomer elementa ');
readln(index);

if index>Count-1 then
begin
writeln('Error: Invalid index');
readln;
CloseFile(F);
Exit;
end;

seek(F, index);

```

```

read(F, Buf);
writeln('element[' , index, ']=', Buf);

CloseFile(F);
readln;
end.

```

## Сортировка слиянием

Рассмотрим еще один алгоритм сортировки – сортировка слиянием. Суть этого алгоритма заключается в следующем: Допустим что имеется два отсортированных файла (массива). Необходимо объединить эти два файла (массива) в один, причем, результирующий файл тоже должен быть отсортированным. Алгоритм сортировки слиянием следующий. Считываем первые элементы списков. Элемент с меньшим значением переносим в результирующий список. Описанный процесс продолжаем до тех пор, пока не исчерпываются элементы в одном из списков. После этого в результирующий список переносятся все оставшиеся в исходном списке элементы. Пример реализации данного алгоритма приведен в программе в листинге 6. Для работы программы необходимо с помощью программы приведенной в листинге 3 создать два файла содержащих отсортированные массивы из целых чисел.

Листинг 34

```

program MergingSort;

{$APPTYPE CONSOLE}
const FileName1='file1.dat';
      FileName2='file2.dat';
      ResultFileName='result.dat';

var F1:File of Integer;      //первый файл
    F2:File of Integer;     //второй файл
    ResFile:File of Integer; //результатирующий файл

    Buf1, Buf2, Buf:Integer;
    Count1, Count2,
    index1, index2:integer;
begin
  AssignFile(F1, FileName1);
  AssignFile(F2, FileName2);
  AssignFile(ResFile, ResultFileName);

  {$I-}
  //открываем файлы с исходными данными
  Reset(F1);
  if IOResult<>0 then Halt(1);

  Reset(F2);
  if IOResult<>0 then
  begin
    CloseFile(F1);
    Halt(1);
  end;

  //создаем результирующий файл
  Rewrite(ResFile);
  if IOResult<>0 then
  begin
    CloseFile(F1);

```

```

    CloseFile(F2);
    Halt(1);
end;
{$I+}

//определяем число компонентов в каждом исходном файле
Count1:=FileSize(F1);
Count2:=FileSize(F2);

index1:=0; //текущая позиция в первом файле
index2:=0; //текущая позиция во втором файле

//слияние двух файлов
read(F1, Buf1);
read(F2, Buf2);

while not((index1>=Count1)or(index2>=Count2)) do
begin
    if Buf1>Buf2 then
    begin
        write(ResFile, Buf2);
        if not EOF(F2) then read(F2, Buf2);
        inc(index2);
    end
    else
    begin
        write(ResFile, Buf1);
        if not EOF(F1) then read(F1, Buf1);
        inc(index1);
    end{if}
end;{while}

//копирование остатка первого файла в результирующий
if index1<Count1 then
begin
    seek(F1, FilePos(F1)-1);
    while not EOF(F1) do
    begin
        read(F1, Buf);
        write(ResFile, Buf);
    end;{while}
end;{if}

//копирование остатка второго файла в результирующий
if index2<Count2 then
begin
    seek(F2, FilePos(F2)-1);
    while not EOF(F2) do
    begin
        read(F2, Buf);
        write(ResFile, Buf);
    end;{while}
end;{if}

//вывод результирующего файла на экран
writeln('Sort Array');
Seek(ResFile, 0); //переходим на начало файла
while not EOF(ResFile) do
begin
    read(ResFile, Buf);

```

```

    write(buf, ' ');
end;

//закрываем все открытые файлы
CloseFile(F1);
CloseFile(F2);
CloseFile(ResFile);
writeln;
writeln('Press Enter to Exit');
readln;
end.

```

## Работа с таблицами

С помощью типизированных файлов удобно организовывать простые базы данных, таблицы, справочники и т.п. Пример реализации простого справочника приведен в ниже приведенной программе. Для этого перепишем программу из лабораторной работы № 5 (листинг 2). Перепишем программу таким образом, чтобы для хранения данных она использовала не массив, а типизированный файл.

Листинг 35

```

program RecFile;

{$APPTYPE CONSOLE}
const TableName='Students.dat';

type TStudent=record
    name    :string[12]; //имя
    familia:string[16]; //фамилия
    grupa  :Integer;    //группа
end;

var F:file of TStudent; //таблица
    Count:integer;      //число строк в таблице
    MenuState:byte;     //код выбранного действия
    i:integer;
    Rec:TStudent;
begin
    AssignFile(F, TableName);
    {$I-}
    Reset(F);
    if IOResult<>0 then
        begin
            Rewrite(F);
            if IOResult<>0 then halt(1);
        end;
    {$I+}

    repeat
        //Выводим меню
        writeln;
        writeln('viberite deistvie');
        writeln('0 - Exit program');
        writeln('1 - Add record');
        writeln('2 - Show all record');
        write('>>'); readln(MenuState);

        //выбираем действие

```

```

case MenuState of
1:
begin //добавление записи в конец массива
writeLn('Insert record >>');

with Rec do
begin
write('name      :'); readLn(name);
write('familia  :'); readLn(familia);
write('gruppa   :'); readLn(gruppa);
end;{with}

//устанавливаем указатель в конец файла
seek(F, FileSize(F)-1);
write(F, Rec);
end;{add}

2:
begin //вывод всех записей на экран
//устанавливаем указатель в начало файла
seek(F, 0);
//рисует таблицу
writeLn('|-----|');

while not EOF(F) do
begin
read(F, Rec);
with Rec do
writeLn(i, ' ', name, ' ', familia, ' ', gruppa);
end;{while}

writeLn('|-----|');
end;{show}
end;{case}
until MenuState=0;

CloseFile(F);
writeLn('Press Enter to exit');
readLn;
end.

```

## Нетипизированные файлы

Третий тип файлов Паскаля, это нетипизированные файлы, этот тип характеризуется тем, что данные имеют не регулярную структуру, например записи различных типов или записи переменной длины, или это просто двоичные данные.

Хотя предполагается работа с двоичными данными неопределенного типа, все равно ведется работа с понятиями запись (блок), размер которой задается при открытии файла, по умолчанию размер записи равен 128 байт. Размер файла должен быть кратным размеру блока, иначе при чтении последней записи возникнет ошибка.

При работе с нетипизированными файлами можно применять все подпрограммы доступные типизированным файлам за исключением процедур Read/Write вместо которых используются BlockRead/BlockWrite.

Таблица 16 – Подпрограммы для работы с нетипизированными файлами

Наименование	Описание
--------------	----------

```
procedure BlockRead(var F: File; var Buf;
Count: Integer [; var AmtTransferred:
Integer]);
```

Считывает указанное число записей из файла F в переменную Buf.

Здесь Buf – буфер для данных считанных с диска; Count – число записей, которые должны быть прочитаны за одно обращение к диску; AmtTransferred – необязательный параметр, содержащий при выходе из процедуры количество фактически обработанных записей.

```
procedure BlockWrite(var f: File; var Buf;
Count: Integer [; var AmtTransferred:
Integer]);
```

Записывает указанное число записей в файл F из переменной Buf.

Здесь Buf – буфер для данных считанных с диска; Count – число записей, которые должны быть прочитаны за одно обращение к диску; AmtTransferred – необязательный параметр, содержащий при выходе из процедуры количество фактически обработанных записей.

Нетипизированный файл объявляется следующим образом:

```
var
```

```
FileVar: file;
```

Функции открытия файла Reset и Rewrite имеют дополнительный параметр, который указывает размер записи, если этот параметр не указан, то используется значение по умолчанию в 128 байт, кстати, это часто является одной из причин для возникновения ошибок, забываем указать этот размер, а при работе считаем, что работаем с байтом. Что бы работать с файлом, как с байтом, надо или установить размер записи в 1 или использовать типизированный файл следующего типа – **file of byte**.

При получении размера файла, результат выдается так же в записях, и если опять же нужно получить размер файла в байтах, также надо устанавливать размер записи в единицу или умножить количество записей на размер записи. Длина файла должна быть кратна длине записи, частичные записи не допустимы.

Ниже приведен пример работы с нетипизированным файлом. Программа осуществляет копирование файла.

Листинг 36

```
program CopyFile;
{$APPTYPE CONSOLE}
var Buf:array[0..1023] of byte;
    F1, F2:file;

    FileName1,
    FileName2:string;

    BytesTransferred:Integer;
    Size:Integer;
begin
  writeln('vvedite filename');
  readln(FileName1);
```



```

//открываем копируемый файл
AssignFile(F1, FileName1);
{$I-}
Reset(F1, 1);
if IOResult<>0 then Halt(1);

//создаем файл копии
writeln('copy file');
readln(FileName2);
AssignFile(F2, FileName2);
Rewrite(F2, 1);
if IOResult<>0 then
begin
  CloseFile(F1);
  Halt(1);
end;
{$I+}

//копируем файл
while not EOF(F1) do
begin
  Size:=SizeOf(Buf);
  BlockRead(F1, Buf, Size, BytesTransferred);
  Size:=BytesTransferred;
  BlockWrite(F2, Buf, Size);
end;

CloseFile(F1);
CloseFile(F2);
end.

```

## Задания к лабораторной работе

1. Наберите программы приведенные в лабораторной работе.
2. Напишите программу для записи двумерного массива в текстовый файл в виде таблицы.
3. Напишите программу осуществляющую чтение заданной строки таблицы из текстового файла и вывод ее на экран. Предусмотреть контроль ошибок.
4. Напишите программу создающую копию заданного текстового файла. Задается имя файла для копирования и имя файла копии.
5. Наберите программу приведенную в листинге и модифицируйте ее добавив следующие функции:
  - поиск записи в файле по имени;
  - вывод на экран всех записей относящихся к заданной группе.

## Вопросы к лабораторной работе

1. Какие функции используются для открытия файлов?
2. Какой алгоритм обработки ошибок ввода-вывода используется в Object Pascal?
3. Какой тип используется для работы с текстовыми файлами? В чем отличие текстовых файлов от других типов файлов?

4. Какие функции могут применяться для работы с текстовыми файлами?
5. Как создать типизированный файл? Чем отличается типизированный файл от других типов файлов?
6. Какие функции допустимы при работе с типизированными файлами?
7. Как создать нетипизированный файл? Какие функции допустимы при работе с нетипизированными файлами?

## **Справочные таблицы**

Таблица 1 – Основные функции для работы с файлами.....	52
Таблица 2 – Коды ошибок ввода-вывода .....	53
Таблица 3 – Управляющие символы .....	54
Таблица 4 – Подпрограммы для работы с текстовыми файлами.....	54
Таблица 5 – Подпрограммы для работы с типизированными файлами.....	57
Таблица 6 – Подпрограммы для работы с нетипизированными файлами.....	63

# Лабораторная работа №6

## Подпрограммы. Рекурсивные подпрограммы.

### Введение

В данной лабораторной работе рассматриваются правила описания подпрограмм в **Object Pascal**. Приведены примеры использования процедур и функций.

Процедуры и функции представляют собой относительно самостоятельные фрагменты программы, именованные и оформленные специальным образом. Отличие процедур от функций заключается в том, что результатом выполнения функции всегда является некоторое значение, поэтому функции можно использовать в выражениях наряду с переменными и константами. Далее будем называть процедуры и функции общим именем – подпрограмма.

### Описание подпрограмм

Подпрограммы описываются перед телом программы основной программы, обычно после блоков **const**, **var**, ...

```
var a, b:integer;  
  
<описание подпрограмм>  
  
begin  
<тело основной программы>  
end.
```

Описание любой подпрограммы состоит из следующих частей:

- заголовок подпрограммы;
- локальные объявления подпрограммы;
- тело подпрограммы.

Обязательными для описания являются первая и последняя часть подпрограммы, а именно: заголовок и тело подпрограммы. Для вызова подпрограммы необходимо указать ее идентификатор (имя), а затем, если необходимо, в круглых скобках список фактических параметров.

```
var c:real;  
  
//описание подпрограммы  
procedure TestProc(  
    a, b:integer; var c:real //список формальных параметров  
);  
  
begin  
    c:=a/b;  
end;  
  
begin  
    TestProc(  
        1, 2, c //вызов подпрограммы  
        // список фактических параметров  
    );  
    writeln(c);  
end.
```

Подробно рассмотрим каждую часть описания подпрограммы.

## Заголовок подпрограммы

Заголовки процедуры и функции описываются следующим образом.

Заголовок процедуры имеет вид:

**procedure** <имя>[(<сп. ф.п.>)];

Здесь <имя> – имя подпрограммы;

<сп.ф.п.> – список формальных параметров.

Заголовок функции в общем аналогичен заголовку процедуры и имеет вид

**function** <имя>[(<сп. ф.п.>)]:<тип>;

Здесь <тип> – тип возвращаемого результата.

Сразу за заголовком процедуры или функции может следовать одна из стандартных директив **assembler**, **external**, **far**, **forward**, **inline**, **interrupt**, **near**. Назначение некоторых директив описано ниже. Кроме того после заголовка процедуры может следовать одна из директив регламентирующих способ передачи списка параметров, например **register**, **pascal**, **stdcall**, ...

В заголовке процедуры или функции может находиться список формальных параметров. Хотя список формальных параметров является не обязательным элементом, но он очень часто применяется при создании подпрограмм. Рассмотрим синтаксис описания списка формальных параметров.

## Список формальных параметров

В общем виде список формальных параметров имеет следующий вид:

<элемент>; <элемент>; ...

где каждый элемент описывается следующим образом

<тип параметра> <id>, <id>, ...:<тип>

где <тип параметра> – способ передачи параметра в подпрограмму (**var**, **const** или **out**);

<id> – идентификатор формального параметра;

<тип> – тип параметра, любой тип Object Pascal.

Любой из формальных параметров может быть либо параметром-значением, либо параметром-переменной, либо параметром-константой.

При вызове подпрограммы список формальных параметров заменяется списком фактических параметров. Список фактических параметров может содержать идентификаторы не совпадающие с идентификаторами списка формальных параметров. Существенны только два ограничения наложенных на список фактических параметров, а именно:

- порядок следования типов списка формальных и списка фактических параметров должны совпадать;
- число формальных и фактических параметров должно совпадать<sup>1</sup>.

Например для следующего описания подпрограммы

---

<sup>1</sup> Исключение составляют умалчиваемые параметры

```
procedure TestProc(a, b:integer; s:string);  
begin  
  ...  
end;
```

Будет корректен следующий вызов

```
TestProc(1, 2, 'hello');
```

и ошибочен следующий

```
TestProc(1, 'hello', 2);
```

Ошибка во втором случае заключается в том, что второй параметр в списке формальных параметров имеет тип `integer`, а в списке фактических тип `string`. Необходимо отметить что Object Pascal для некоторых типов выполняет автоматическое преобразование типов. Так например тип `integer` автоматически может быть преобразован в необходимый вещественный тип `real`, `double`, ...

Рассмотрим подробнее некоторые типы формальных параметров.

### Параметры-значения

Если тип формального параметра не указан, то параметр считается параметром-значением. Особенностью параметров-значений является то, что в подпрограмму передается не сам параметр, а его копия. Параметры-значения нечувствительны к изменению их внутри подпрограммы, т.е. если параметр будет изменен внутри подпрограммы, то он останется неизменным вне подпрограммы.

```
procedure TestProc(a, b, c:integer);
```

В качестве фактического параметра при вызове подпрограммы можно использовать константу или переменную, например

```
TestProc(1, 2, 3);
```

или

```
TestProc(1, a, c);
```

### Параметры-переменные

Для определения параметра как параметра-переменной необходимо использовать зарезервированное слово `var`, например:

```
procedure TestProc(var a, b, c:integer);
```

В отличие от параметров-значений параметры-переменные передаются внутрь подпрограммы. Таким образом при изменении такого параметра в теле подпрограммы он изменяется и вне подпрограммы. Отсюда следует область применения таких параметров – их применяют в случае когда необходимо получить результат работы подпрограммы. Кроме того следует иметь ввиду, что в качестве фактических параметров-переменных подпрограммы можно использовать только переменные.

```

var res:integer;

procedure MulProc(a, b:integer; var res:integer);
begin
  res:=a*b; //изменение параметра-переменной
end;

begin
  res:=1;
  MulProc(2, 3, res);
  writeln(res);
end.

```

В результате выполнения выше указанной программы на экран будет выведено число 6.

Если определить формальный параметр `res` как параметр значение (удалить `var` перед параметром `res`), то на экран будет выведено число 1, т. к. в подпрограмму `MulProc` будет передан не фактический параметр `res`, а его копия которая будет утеряна при выходе из подпрограммы.

## Параметры-константы

При передаче параметров как параметров-переменных фактически передается адрес переменной, для большинства современных персональных компьютеров размер адреса составляет 4 байта. Как уже говорилось выше при использовании параметров-значений в подпрограмму передаются копии фактических параметров. Если размер фактических параметров велик и число вызовов подпрограммы велико, то постоянное создание копий фактических параметров может потребовать значительных ресурсов компьютера. Очевидно, что если параметры не планируется изменять в теле программы, то рациональнее, с точки зрения производительности, было бы передавать «большие параметры» как параметры-переменные, но существует вероятность неправильного использования таких параметров, например случайного изменения в результате написания некорректного кода.

Для увеличения надежности подпрограмм были введены параметры-константы.

Механизм передачи таких параметров такой же как и параметров-переменных, т. е. передается адрес переменной, но компилятор строго следит за тем чтобы внутри

подпрограммы данный параметр не изменялся. Для определения параметра-константы необходимо использовать зарезервированное слово **const**, например

```
procedure TestProc(const a:integer; b, c:integer);
```

Если в теле подпрограммы присвоить параметру a какое либо значение, то компилятор выдаст ошибку, например

```
var res:integer;  
procedure TestProc(const a:integer; b, c:integer);  
begin  
  a:=c*b; //здесь компилятор выдаст ошибку  
end;  
...
```

Следует также отметить, что в качестве фактических параметров-констант в отличие от параметров-переменных допустимо использовать как переменные так и константы, например следующие два обращения к подпрограмме будут корректными.

```
var res:integer;  
...  
TestProc(res, 1, 2);  
TestProc(1, 2, 3);
```

## Нетипизированные параметры

Параметр считается нетипизированным если его тип не указан в заголовке подпрограммы. Нетипизированными могут быть только параметры переменные. Например:

```
procedure SendBuffer(var Buf);
```

Нетипизированные параметры используются в случае если тип данных передаваемых в подпрограмму не существен.

## Умалчиваемые параметры

Умалчиваемые параметры<sup>2</sup> это параметры которые могут опускаться при обращении к подпрограмме. Умалчиваемые параметры всегда замыкают список формальных параметров и имеют вид

<id>: <тип> = <значение>

Например

```
procedure Rectangle(x1, y1, x2, y2:Integer; Color:Byte=c1Black);
```

При вызове этой процедуры допустимы оба ниже приведенных варианта

```
Rectangle(0, 0, 100, 50);  
Rectangle(0, 0, 100, 50, c1Blue);
```

Если в подпрограмме используется два и более параметров по умолчанию, то следует указывать все параметры вплоть до последнего опускаемого, например

<sup>2</sup> Использование умалчиваемых параметров возможно начиная с Delphi 4.

```
procedure Rectangle(x1, y1, x2, y2:Integer;Color:Byte=c1Black;  
Filled:Boolean=False);
```

Будут допустимы следующие обращения

```
Rectangle(0, 0, 100, 50);  
Rectangle(0, 0, 100, 50, c1Blue);  
Rectangle(0, 0, 100, 50, c1Blue, True);
```

Но не допустимо следующее

```
Rectangle(0, 0, 100, 50, True);
```

## Параметры-массивы

При передаче в качестве формальных параметров массивов существуют определенные ограничения. Для передачи массива в качестве формального параметра сначала необходимо объявить соответствующий тип, а затем указать в списке формальных параметров параметр созданного типа, например

```
type TVector=array [1..3] of real;  
function DotProduct(AVector:TVector):real;
```

Кроме того, **Delphi** поддерживает так называемые **открытые массивы**, которые служат для разрешения проблемы передачи одномерных массивов. Открытый массив представляет собой формальный параметр подпрограммы и имеет следующий синтаксис

<id>: **array of** <тип>

Внутри подпрограммы такой массив трактуется как одномерный массив с нулевой нижней границей. Верхняя граница может быть определена с помощью функции High. Например выше описанную функцию можно определить как:

```
function DotProduct(AVector: array of real):real;
```



**Совет:** Рекомендуется всегда вместо передачи массива как параметра-значения передавать его как параметр-константу, что позволяет сократить число обращений к оперативной памяти и существенно увеличить производительность программы. Например выше приведенную функцию лучше переписать следующим образом

```
function DotProduct(const AVector: array of real):real;
```

## Параметры-строки

Так как параметр типа ShortString по сути является массивом символов, то передача такого параметра в подпрограмму осуществляется аналогично массивам, например

```
type NameStr=string[15];  
procedure ShowName(AName: NameStr);
```

Строки типа AnsiString передаются как обычные параметры, например

```
procedure ShowName(AName: String);
```



**Совет:** В обоих выше приведенных случаях строки, как и массивы, рекомендуется передавать как параметры-константы. Не смотря на то, что переменная типа AnsiString по сути не содержит саму строку, а лишь адрес памяти где она располагается, компилятор все равно при передаче такого параметра в подпрограмму



создаст копию строки. Именно по этому оптимальнее использовать параметр-константу, а не параметр-значение, например

```
type NameStr=string[15];  
  
procedure ShowName(const AName: NameStr);  
  
или  
  
procedure ShowName(const AName: String);
```

## Параметры-записи

В качестве формальных параметров могут использоваться переменные и константы типа запись. Правила объявления таких параметров схожи с объявлением параметров-массивов. Кроме того, для таких параметров лучше использовать параметры-константы вместо параметров-значений, например

```
type TPerson=record  
    name:string[15];  
    fam :string[15];  
    oth :string[23];  
end;  
  
procedure ShowInfo(const APerson:TPerson);
```

## Локальные элементы подпрограмм

Локальные элементы являются необязательными при описании подпрограмм. Они располагаются между заголовком подпрограммы и телом подпрограммы. В качестве локальных элементов могут использоваться переменные, константы, типы, метки, а также другие подпрограммы, например:

```
procedure TestProc(var a:integer; b:real);  
  
    type TVector=array [1..3] of real;  
    var i:integer;
```

В примере приведены два локальных описания. Локальный тип и локальная переменная. Обратите внимание, что недопустимо совпадение идентификаторов формальных параметров и идентификаторов локальных элементов подпрограмм (переменных, констант и т.п.). Особенностью локальных элементов является ограниченное время существования и ограниченная область видимости.

Время существования локальных элементов подпрограммы ограничено временем выполнения тела подпрограммы. После выхода из подпрограммы все локальные элементы уничтожаются.

Обратите внимание что значение локальных переменных после выхода из подпрограммы теряется и при следующем вызове данной подпрограммы локальные переменные содержат некоторые случайные значения. Поэтому перед их использованием им необходимо присвоить нужные значения.

Область видимости локальных элементов ограничена подпрограммой, включая вложенные подпрограммы, т. е. обращение к любым локальным элементам подпрограмм разрешено в теле подпрограммы и во всех вложенных подпрограммах и запрещено вне данной подпрограммы.

Типичная структура программы использующей подпрограммы имеет следующий вид

```
program ManyProc;  
  
procedure A;  
    procedure A1;
```

```

.....
begin
end; {A1}

procedure A2;
.....
begin
end; {A2}

begin
end; {A}

procedure B;
  procedure B1;
.....
begin
end; {B1}

  procedure B2;
    procedure B21;
      begin
        end; {B21}

    procedure B22;
      begin
        end; {B22}

    begin
      end; {B2}

  begin
    end; {B}

begin
end. {ManyProc}

```

Структуру выше приведенной программы можно представить в виде следующей схемы

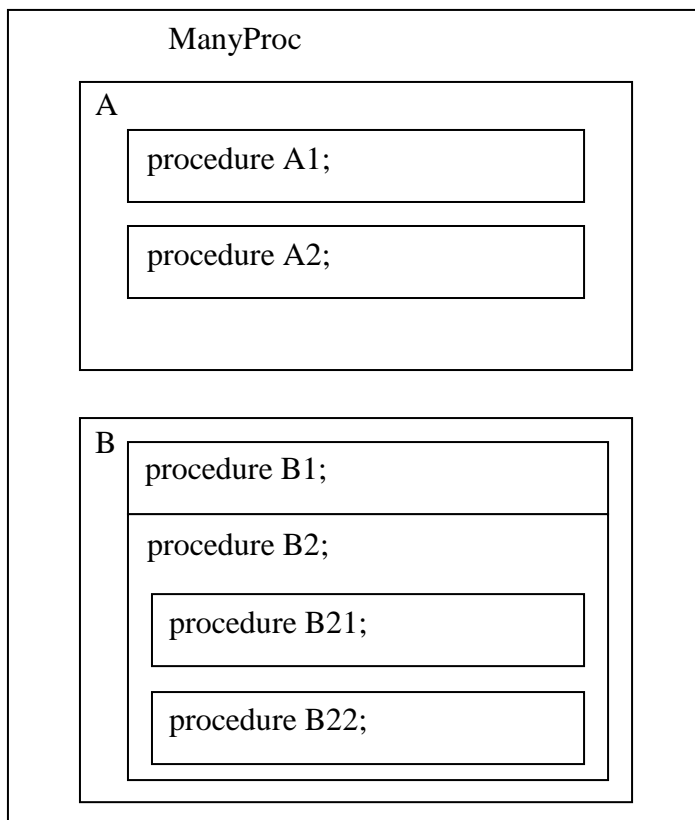


Рисунок 5 – Структура программы

## Тело подпрограммы

Тело подпрограммы находится в операторных скобках следующих за заголовком подпрограммы.

```

begin
<тело подпрограммы>
end.

```

Тело подпрограммы может содержать выражения и любые операторы Object Pascal, в том числе и вызовы других подпрограмм.

При описании тела функции существует одна особенность. Так как функция может использоваться в выражениях, то необходимо указать результат вычисления функции который следует использовать в выражении. Сделать это можно двумя способами:

- в теле функции должна присутствовать строка вида <имя функции>:=<результат>;
- в теле функции должна присутствовать строка вида Result:=<результат>.

Следует отметить, что после указанных строк выполнение функции не прерывается. Практически, имя функции (или идентификатор Result) рассматривается как формальный параметр-переменная и соответственно к нему применимы все правила для работы с переменными, например

```

function mul(a, b:real):real;
begin
  mul:=a*b;
end;

```



**Совет:** Способ указания результата вычисления функции приведенный выше традиционно используется в Turbo Pascal. В Object Pascal принят другой, более удобный

способ, а именно использование параметра *Result*. Преимуществом использования параметра *Result* является то что при изменении имени функции нет необходимости изменять тело функции, например

```
function mul(a, b:real):real;  
begin  
  Result:=a*b;  
end;
```

## Рекурсивные подпрограммы и опережающее описание

Рекурсивной подпрограммой называют подпрограмму которая в своем теле прямо или косвенно вызывает саму себя. Рассмотрим подпрограмму вычисляющую факториал числа<sup>3</sup>. Факториал числа вычисляется по формуле

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

Листинг 37

```
program Factorial;  
  
{ $APPTYPE CONSOLE }  
function Fact(n:Integer):integer;  
begin  
  if (n=0) or (n=1) then Result:=1  
  else Result:=n*Fact(n-1);  
end;  
  
begin  
  writeln(Fact(10));  
  readln;  
end.
```

В общем случае рекурсия имеет следующий вид

```
procedure First;  
begin  
  ...  
  Second;  
  ...  
end;  
  
procedure Second;  
begin  
  ...  
  First;  
  ...  
end;
```

<sup>3</sup> Рекурсивный алгоритм вычисления факториала не эффективен и рассматривается лишь в качестве простого примера использования рекурсивных подпрограмм.

Из выше приведенного листинга видно, что процедура First вызывает процедуру Second, а процедура Second вызывает процедуру First. Если попытаться откомпилировать выше приведенный фрагмент программы, то компилятор выдаст ошибку, «Идентификатор Second не определен». Как известно все идентификаторы в **Object Pascal** обязательно должны быть описаны перед первым использованием. Как видно из листинга ни какой порядок описания процедур не может удовлетворить этому условию. Для выхода из этой ситуации применяется **опережающее описание подпрограмм**. Для этого необходимо описать заголовок одной из подпрограмм с директивой **forward**. Перепишем приведенный выше пример

```
procedure Second; forward;

procedure First;
begin
  ...
  Second;
  ...
end;

procedure Second;
begin
  ...
  First;
  ...
end;
```

Этот фрагмент откомпилируется без ошибок, т.к. при вызове процедуры First идентификатор процедуры Second уже будет описан.

## Примеры использования подпрограмм

В данной главе приведены некоторые приемы использования подпрограмм.

### **Вычисление значений выражений**

Листинг 38

```
program Func;
{$APPTYPE CONSOLE}
function MyFunc(x:real):real;
begin
  if x>0 then result:=x
  else result:=x*x;
end;
var x:real;
begin
  write('x=');
  readln(x);
  writeln('F(x)=', 7*x*MyFunc(x):6:3);
  readln;
end.
```

## Ввод-вывод переменных типа запись

Как говорилось выше в **Object Pascal** нет стандартных средств для операций ввода-вывода для переменных типа запись. Поэтому удобно написать собственные подпрограммы для осуществления этих операций. Пример таких подпрограмм приведен ниже.

Листинг 39

```
program Records;

{$APPTYPE CONSOLE}
uses SysUtils;

type TPerson=record
    name:string[12];
    fam :string[16];
    otch:string[20];
    Birthday:TDateTime;
end;

//ввод записи
procedure InRecord(var Person:TPerson);
var s:string;
begin
    with Person do
        begin
            write('name      ');
            readln(name);
            write('familia  ');
            readln(fam);
            write('othectvo ');
            readln(otch);
            write('Birthday ');
            readln(s);

            //преобразуем строку в тип дата-время
            Birthday:=StrToDate(s);
        end;{with}
    end;{InRecord}

//вывод записи
procedure OutRecord(const Person:TPerson);
begin
    with Person do
        begin
            writeln('name      ', name);
            writeln('familia  ', fam);
            writeln('othectvo ', otch);
            writeln('Birthday ', DateToStr(Birthday));
        end;{with}
    end;{OutRecord}

//основная программа
var P:TPerson;
begin
    InRecord(P);
    writeln('_____');
    OutRecord(P);
    readln;
end.
```

Рассмотрим еще один пример по работе с записями. Перепишем программу из лабораторной работы №5 (листинг 2) с использованием подпрограмм.

Листинг 40

```
program TableRec;

{$APPTYPE CONSOLE}
const n=10; //емкость массива
type TStudent=record
    name :string[12]; //имя
    familia:string[16]; //фамилия
    группа :Integer; //группа
end;

var Table:array [1..n] of TStudent; //таблица
    Count:integer; //число строк в таблице
    MenuState:byte; //код выбранного действия

//выводит меню, возвращает результат выбора
function ShowMenu:Integer;
begin
    writeln;
    writeln('Viberite deistvie');
    writeln('0 - Exit program');
    writeln('1 - Add record');
    writeln('2 - Show all record');
    write('>>');
    readln(Result);
end;

//ВВОД ЗАПИСИ
procedure InRecord(var P:TStudent);
begin
    writeln('Insert record >>');
    with P do
    begin
        write('name :'); readln(name);
        write('familia :'); readln(familia);
        write('gruppa :'); readln(gruppa);
    end; {with}
end; {InRecord}

//ВЫВОД ЗАПИСИ
procedure ShowRecord(const P:TStudent);
begin
    with P do writeln(name, ' ', familia, ' ', gruppa);
end; {ShowRecord}

//добавление записи в конец массива
procedure AddRecord;
begin
    //если число элементов превышает емкость массива
    if Count=n then
        writeln('Error: array overflow')
    else //если в массиве есть место, то добавляем запись
    begin
        inc(Count);
        InRecord(Table[Count]);
    end; {else}
end; {AddRecord}
```

```

//печать таблицы
procedure ShowAllRecord;
var i:integer;
begin
  //рисуем таблицу
  writeln('|-----|');
  for i:=1 to Count do
    begin
      write(i, ' ');
      ShowRecord(Table[i]);
    end;
  writeln('|-----|')
end;

//основная программа
begin
  Count:=0;

  repeat
    //выводим меню
    MenuState:=ShowMenu;

    //выбираем действие
    case MenuState of
      1: AddRecord;
      2: ShowAllRecord;
    end;{case}
  until MenuState=0;

  writeln('Press Enter to exit');
  readln;
end.

```

## ***Работа с массивами***

В листинге приведенном ниже реализованы некоторые простые функции для работы с массивами.

**Листинг 41**

```

program Arrays;

{$APPTYPE CONSOLE}

//ввод массива
procedure InArray(var a:array of real);
var i:Integer;
begin
  for i:=0 to High(a) do
    begin
      write('a[', i, ']=');
      readln(a[i]);
    end;{for}
  end;{InArray}

//вывод массива
procedure OutArray(const a:array of real);
var i:Integer;
begin
  for i:=0 to High(a) do
    writeln('a[', i, ']=', a[i]:6:3);
  end;

```



```

//вычисление максимального элемента массива
function max(const a:array of real):real;
var i:Integer;
begin
  result:=a[0];
  for i:=1 to High(a) do
    if result<a[i] then result:=a[i];
  end;

//вычисление минимального элемента массива
function min(const a:array of real):real;
var i:Integer;
begin
  result:=a[0];
  for i:=1 to High(a) do
    if result>a[i] then result:=a[i];
  end;

//вычисление среднего элемента массива
function avg(const a:array of real):real;
var i:Integer;
    sum:real;
begin
  sum:=0;
  for i:=0 to High(a) do
    sum:=sum+a[i];
  result:=sum/(High(a)+1);
end;

//основная программа
const n=5;
var Data:array [1..n] of real;
begin
  InArray(Data);
  writeln('min=', min(Data):6:3);
  writeln('max=', max(Data):6:3);
  writeln('avg=', avg(Data):6:3);
  OutArray(Data);
  readln;
end.

```

## Сортировка методом прочесывания

Продолжим рассматривать алгоритмы сортировки массивов. Рассмотрим алгоритм сортировки методом прочесывания. По сути алгоритм улучшает метод пузырьковой сортировки, а именно меняются местами не два соседних элемента, а два элемента отстоящих друг от друга на некоторое расстояние. При каждом проходе расстояние между элементами уменьшается до тех пор пока не достигнет единицы в этом случае имеет место пузырьковая сортировка. Процесс останавливается после выполнения двух условий: расстояние между элементами стало равным единице и за проход не было выполнено ни одной перестановки элементов. Экспериментально получены следующие оптимальные параметры алгоритма: на каждом проходе необходимо уменьшать расстояние между элементами в 1.3 раза, расстояния 9 и 10 являются не оптимальными.

Пример реализации сортировки методом прочесывания приведен в листинге 6.

**Листинг 42**

```
program CombSort;
```

```

{$APPTYPE CONSOLE}
const n=10;
var a:array [1..n] of real=(1, 5, 2, 7, 45, 12, 3, 4, 10, 89);
    i:integer;

//вывод массива
procedure OutArray(const a:array of real);
var i:Integer;
begin
    for i:=0 to High(a) do
        writeln('a[' , i, ']=' , a[i]:6:3);
    end;

//сортировка массива методом прочесывания
procedure CompSortProc(var Data: array of real);
var Done:Boolean;
    Gap :Integer;
    i, j:Integer;
    buf :real;

begin
    {начинаем сортировку с расстояния равного количеству
элементов}
    Gap:=High(Data)+1;

    repeat
        //устанавливаем флаг перестановки
        Done:=True;

        //вычисляем новый интервал
        Gap:=Trunc(Gap/1.3);
        if Gap<1 then Gap:=1
        else if (Gap=9) or (Gap=10) then Gap:=11;

        {упорядочить два элемента отстоящих друг от друга на Gap
элементов}
        for i:=0 to High(a)-Gap do
            begin
                j:=i+Gap;
                if a[i]>a[j] then //меняем элементы местами
                    begin
                        buf:=a[i];
                        a[i]:=a[j];
                        a[j]:=buf;
                        Done:=False; //сбрасываем флаг перестановки
                    end;{if}
            end;{for i}
        until (Done)and(Gap=1);
    end;{CompSortProc}

//основная программа
begin
    CompSortProc(a);
    OutArray(a);
    readln;
end.

```

## Процедурные типы

Процедурные типы используются как средство передачи процедур и функций в качестве фактических параметров при обращении к другим подпрограммам.

Для объявления процедурного типа необходимо описать заголовок подпрограммы процедуры или функции без указания ее имени, например

```
type MyProc=procedure;  
      PrintProc=procedure (a, b:Integer);  
      Func=function(x:real):real;
```

Как видно из приведенных примеров, существует два вида процедурных типа: тип-процедура и тип функция.

В качестве примера рассмотрим подпрограмму вычисления значения определенного интеграла методом прямоугольников.

Метод вычисления значения определенного интеграла методом прямоугольников использует геометрический смысл определенного интеграла, который заключается в том, что определенный интеграл – это площадь под кривой на отрезке интегрирования.

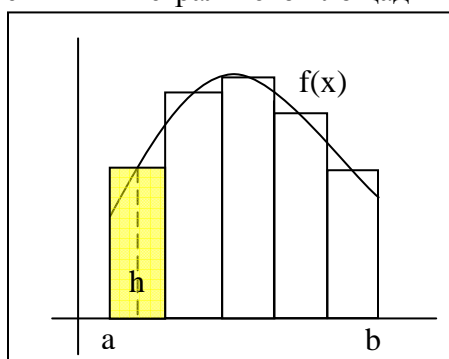


Рисунок 7

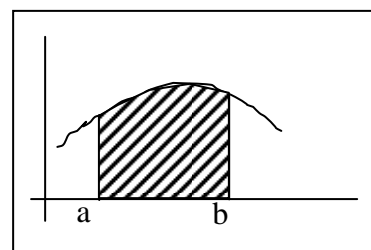


Рисунок 6

Весь интервал интегрирования разбивается на равные отрезки. На каждом отрезке подынтегральная функция заменяется ступенчатой функцией значение которой равно значению исходной функции в середине отрезка. Площадь каждого прямоугольника вычисляется по формуле

$$s_i = h \cdot f(a + h(1/2 + i)).$$

где  $h = (b - a)/n$ ,  $n$  – число точек разбиения.

Следовательно значение определенного интеграла можно вычислить по формуле

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} s_i$$

Значение интеграла вычисляется тем точнее, чем меньше шаг интегрирования

Листинг программы для вычисления значения интеграла методом прямоугольников приведен в листинге

Листинг 43

```
program Integral;  
  
{ $APPTYPE CONSOLE }  
type Func=function(x:real):real;  
  
function Int(F:Func; //подынтегральная функция  
             a, b:real; //пределы интегрирования  
             n:integer //число точек разбиения
```

```

    ):real;
var i:Integer;
    h, s:real;
begin
    h:=(b-a)/n; //шаг интегрирования
    s:=0;
    for i:=0 to n-1 do
        begin
            s:=s+h*F(a+h/2+h*i);
        end;
    Result:=s;
end;
//линейная функция
function Lin(x:real):real;
begin
    result:=x;
end;
//квадратичная функция
function MySqr(x:real):real;
begin
    result:=x*x;
end;
begin
    writeln('Lin    ', Int(Lin, 0, 1, 10):6:5);
    writeln('Sqr    ', Int(MySqr, 0, 1, 10):6:5);
    readln;
end.

```

Как видно из листинга типу Func соответствует любая функция имеющая тот же список параметров (число и тип параметров).

## Задания к лабораторной работе

1. Наберите программы приведенные в лабораторной работе.
2. Составьте программу для вычисления значений следующих функций:

$$f(x) = \begin{cases} x < 0, 2x \\ 0 \leq x < 3, x^3 \\ x \geq 3, \ln(x) \end{cases} \quad f(x) = \begin{cases} x > 0, x^2 \\ x \leq 0, x^3 + 2x \end{cases}$$

3. Перепишите задание 2 из лабораторной работы №5 с использованием подпрограмм. В качестве примера используйте программу из листинга 4
4. Оформите программы для поиска и сортировки массивов из лабораторной работы №4 в виде подпрограмм.
5. Составьте подпрограммы для ввода (вывода) данных в (из) двумерного массива.
6. Оформите программы для вставки и удаления элементов массивов из лабораторной работы №4 в виде подпрограмм.
7. Составьте подпрограмму для печати таблицы значений функции. В качестве параметров подпрограммы служат табулируемая функция, интервал табулирования и число точек.

## Вопросы к лабораторной работе

1. Какие виды подпрограмм существуют в Object Pascal, как они описываются?
2. Из каких частей состоит подпрограмма?
3. Как связаны между собой список формальных параметров и список фактических параметров?
4. Чем отличаются и как описываются параметры-значения, параметры-переменные, параметры-константы?
5. Что такое открытые массивы, когда они применяются?
6. Как передать в подпрограмму многомерный массив?
7. Что такое область видимости локальной переменной?
8. Какое время жизни имеет локальная переменная?
9. Как оформляется передача результата вычисления функции?
10. Что такое рекурсия?
11. Зачем нужно опережающее описание подпрограмм?
12. Как объявить процедурный тип?
13. Объясните алгоритм работы метода сортировки прочесыванием.
14. Что такое и как описываются умалчиваемые параметры?

## Справочные таблицы

# Лабораторная работа №7

## Модули. Файл проекта.

### Введение

В данной лабораторной работе рассмотрены модули Object Pascal, назначение каждой секции модуля, приведен пример модуля. Также рассмотрены некоторые стандартные модули Object Pascal.

Модуль – это автономно компилируемая программная единица, включающая в себя различные компоненты интерфейсного раздела (типы, константы, переменные, подпрограммы) и, возможно, некоторые операторы иницилирующего раздела. Объекты описанные в интерфейсной части доступны для других модулей и основной программы. Тела процедур и функций располагаются в разделе реализации модуля и скрыты от пользователя.

### Структура модуля

Модуль в Object Pascal имеет следующую структуру:

```
unit <имя_модуля>;  
interface  
<раздел интерфейса>  
implementation  
<раздел реализации>  
initialization  
<раздел инициализации>  
finalization  
<раздел деинициализации>  
end.
```

Рассмотрим подробнее назначение каждого элемента модуля.

- <имя\_модуля> – правильный идентификатор **Object Pascal**, заголовок модуля;
- interface** – зарезервированное слово, начинает раздел интерфейса модуля;
- implementation** – зарезервированное слово, начинает раздел реализации модуля;
- initialization** – зарезервированное слово, начинает раздел инициализации модуля;
- finalization** – зарезервированное слово, начинает раздел деинициализации модуля;
- end** – зарезервированное слово, признак конца модуля.

Таким образом модуль состоит из заголовка и четырех разделов. Обязательным является только раздел интерфейса, остальные разделы могут отсутствовать.

### Заголовок модуля

Заголовок модуля состоит из зарезервированного слова **unit** и следующего за ним имени модуля. Имя модуля – правильный идентификатор Object Pascal, совпадающий с

именем файла модуля. Например если модуль имеет имя Unit1, то имя файла модуля Unit1.pas.

## Подключение модулей

Для подключения модуля к основной программе или к другому модулю необходимо использовать следующую конструкцию

```
uses <ИМЯ_МОДУЛЯ>, < ИМЯ_МОДУЛЯ >, ...;
```

где **uses** – зарезервированное слово.

Объявление **uses** может следовать за зарезервированными словами **interface**, **implementation** в модулях и после имени программы в основной программе, например

Листинг 44 – Подключение модуля к другому модулю



```
unit MyUnit;  
interface  
uses Windows, MySecondUnit;  
  
<раздел интерфейса>  
  
implementation  
  
<раздел реализации>  
end.
```

Листинг 45 – Подключение модуля к основной программе

```
program TestPrg;  
uses Windows, MySecondUnit;  
  
begin  
  
end.
```

Рассмотрим еще один важный вопрос: «Где компилятор ищет файлы модулей?». Порядок поиска файлов модулей следующий:

1. осуществляется поиск в каталоге основной программы;
2. далее просматриваются каталоги для поиска указанные в настройках среды;
3. далее просматривается каталог Delphi7\Lib.

Для установки списка каталогов для поиска модуля необходимо вызвать диалоговое окно  Options... из меню Project. На вкладке Directories/Conditionals необходимо нажать кнопку  на против строки Search path, появится окно ввода списка каталогов для поиска в которое необходимо ввести список каталогов для поиска модулей (рисунок 1).

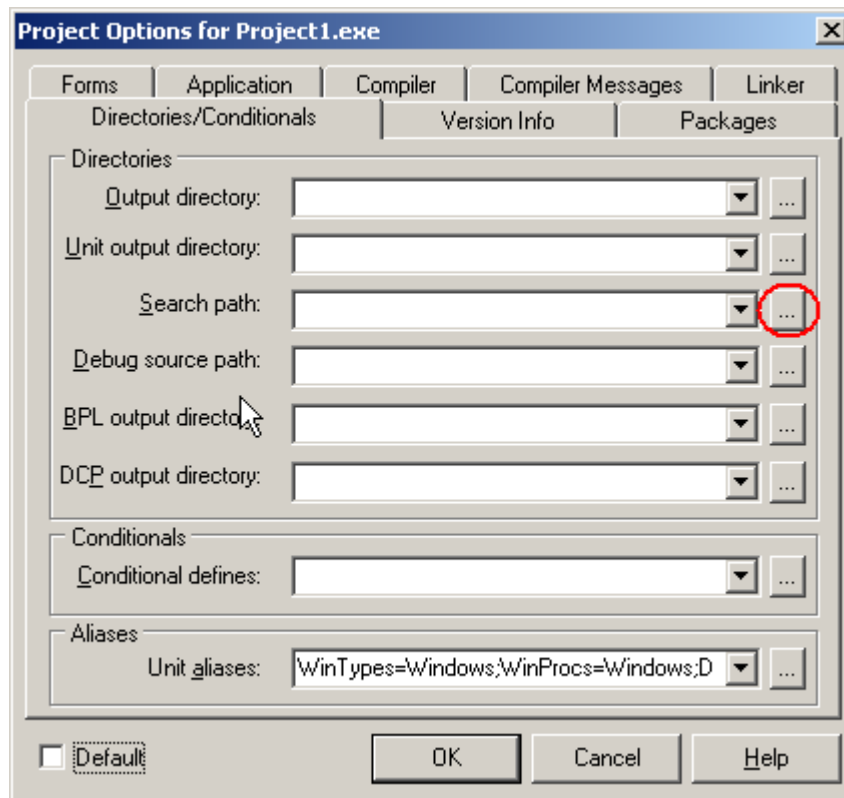


Рисунок 8 – Окно настроек проекта (раздел каталоги)

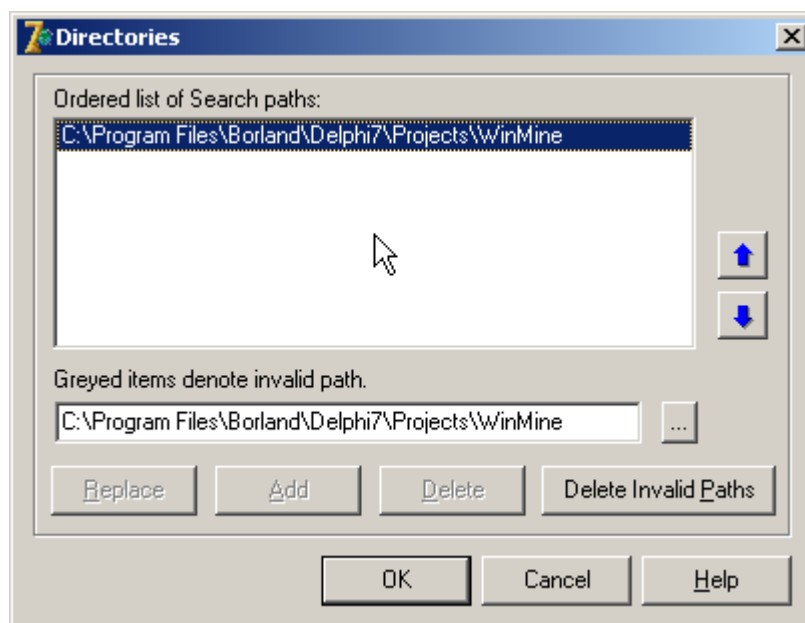


Рисунок 9 – Окно ввода списка каталогов

## Создание модулей

Для создания нового модуля необходимо в выполнить команду **File>New>Unit** (рисунок 3)



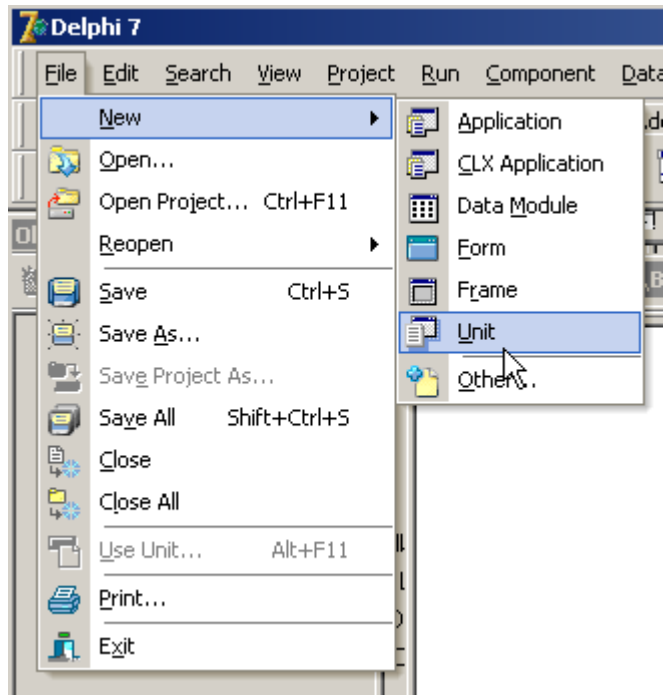


Рисунок 10 – Создание нового модуля

Появится окно с заготовкой модуля (рисунок 4).

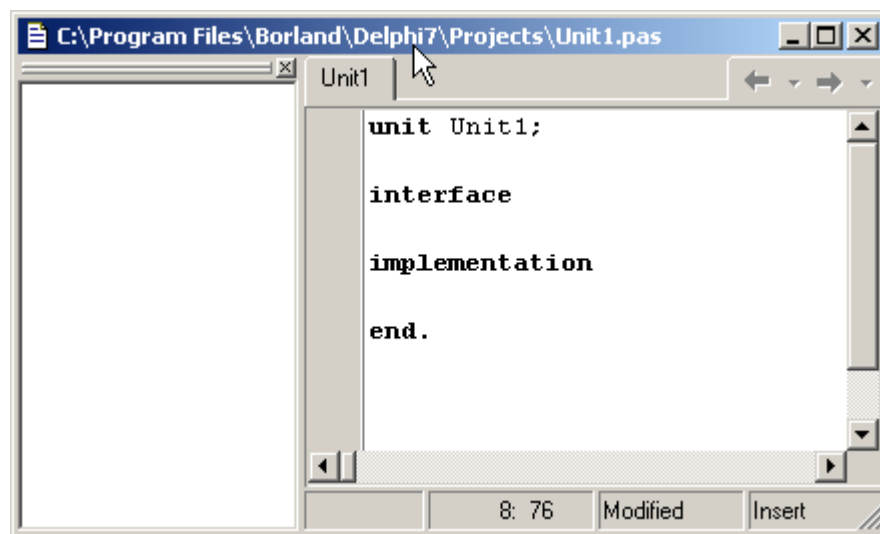


Рисунок 11 – Новый модуль

В редакторе может быть открыто сразу несколько модулей, например

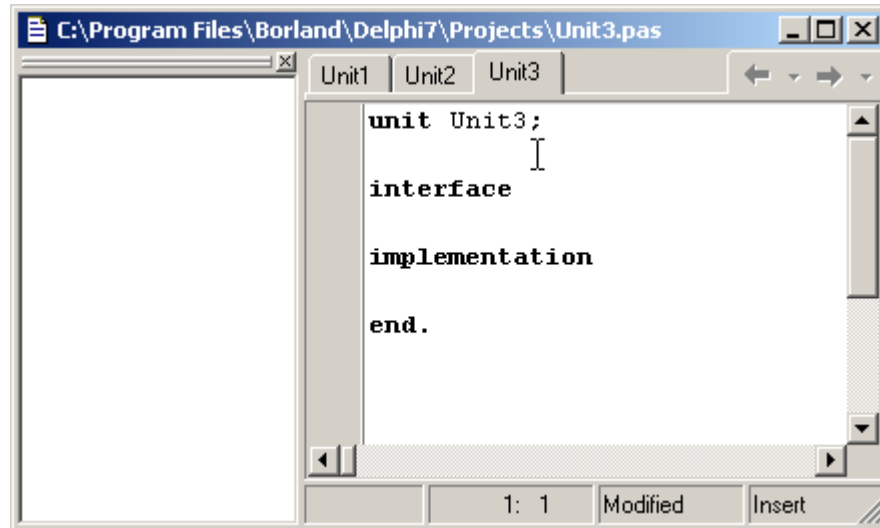




Рисунок 12 – В редакторе кода открыты несколько модулей

Для переключения между модулями достаточно выбрать соответствующую закладку. Чтобы сохранить только редактируемый модуль необходимо нажать кнопку Save  на панели инструментов или Ctrl+S на клавиатуре. Чтобы сохранить все открытые модули необходимо нажать кнопку SaveAll  на панели инструментов или Shift+Ctrl+S на клавиатуре.

После компиляции модуля создается файл с расширением dcu содержащий машинный код модуля программы. Откомпилировать модуль можно только в составе проекта. При компиляции проекта компилятор сначала ищет откомпилированные файлы модулей и, если они отсутствуют, то сначала компилирует исходные файлы модулей в dcu-файлы, а затем «собирает» приложение используя данные откомпилированных модулей.

Для удачной перекомпиляции проекта не требуются исходные файлы модулей (pas), а достаточно лишь откомпилированных файлов модулей (dcu). Следует иметь ввиду, что внести изменения в откомпилированные файлы модулей невозможно. Поэтому если возможность внесения изменений в текст модуля является важной, то необходимо также сохранять исходные файлы модулей. Преимуществом использования откомпилированных файлов модулей является то, что возможна поставка библиотек функций без раскрытия исходного кода, что часто применяется при коммерческой поставке библиотек подпрограмм.

## Раздел интерфейса

Раздел интерфейса модуля открывается зарезервированным словом **interface**. В этом разделе содержатся объявления всех глобальных объектов модуля (типов, констант, переменных и подпрограмм), которые должны стать доступными основной программе и/или другим модулям. При объявлении глобальных подпрограмм в интерфейсной части указывается только их заголовок, например:

```

unit Tables;

interface
type TStudent=record
    name    :string[12]; //имя
    familia:string[16]; //фамилия
    группа :Integer; //группа
end;

//выводит меню, возвращает результат выбора
function ShowMenu:Integer;

```

```
//добавление записи в конец массива
procedure AddRecord;

//печатать таблицы
procedure ShowAllRecord;
```

Если включить этот модуль в состав предложения **uses** в другом модуле или программе, то в них будут доступны тип TStudent и перечисленные подпрограммы.

## **Раздел реализации**

Раздел реализации начинается зарезервированным словом **implementation** и содержит обязательно описания подпрограмм, объявленных в интерфейсной части. В ней могут объявляться локальные для модуля объекты – вспомогательные типы, константы, переменные и подпрограммы, а также метки, если они используются в разделе инициализации.

Описанию подпрограммы, объявленной в интерфейсной части модуля, в исполняемой части должен предшествовать заголовок, в котором можно опускать список формальных параметров (и тип результата для функции), так как они уже описаны в интерфейсной части. Но если заголовок подпрограммы приводится в полном виде, т.е. со списком формальных параметров и объявлением результата, он должен совпадать с заголовком, объявленным в интерфейсной части, например:

```
unit Tables;

interface
type TStudent=record
    name :string[12]; //имя
    familia:string[16]; //фамилия
    grupper :Integer; //группа
end;

//выводит меню, возвращает результат выбора
function ShowMenu:Integer;

//добавление записи в конец массива
procedure AddRecord;

//печатать таблицы
procedure ShowAllRecord;

implementation
const n=10; //емкость массива

var Table:array [1..n] of TStudent; //таблица
    Count:integer; //число строк в таблице

function ShowMenu:Integer;
begin
    writeln;
    writeln('viberite deistvie');
    writeln('0 - Exit program');
    writeln('1 - Add record');
    writeln('2 - Show all record');
    write('>>');
    readln(Result);
end;
```

```

//ВВОД записи
procedure InRecord(var P:TStudent);
begin
  writeln('Insert record >>');
  with P do
    begin
      write('name      :'); readln(name);
      write('familia  :'); readln(familia);
      write('gruppa   :'); readln(gruppa);
    end;{with}
  end;{InRecord}

//Вывод записи
procedure ShowRecord(const P:TStudent);
begin
  with P do writeln(name, ' ', familia, ' ', gruppa);
end;{ShowRecord}

//добавление записи в конец массива
procedure AddRecord;
begin
  //если число элементов превышает емкость массива
  if Count=n then
    writeln('Error: array overflow')
  else //если в массиве есть место, то добавляем запись
    begin
      inc(Count);
      InRecord(Table[Count]);
    end;{else}
  end;{AddRecord}

//печать таблицы
procedure ShowAllRecord;
var i:integer;
begin
  //рисуем таблицу
  writeln('|-----|');
  for i:=1 to Count do
    begin
      write(i, ' ');
      ShowRecord(Table[i]);
    end;
  writeln('|-----|')
end;

initialization
  Count:=0;
end.

```

Объявления сделанные в разделе реализации модуля будут недоступны из других модулей. Так, например, из другого модуля нельзя будет обратиться к переменной Table.

### ***Раздел инициализации***

В разделе инициализации размещаются исполняемые операторы, содержащие некоторый фрагмент программы. Эти операторы исполняются до передачи управления основной программе и обычно используются для подготовки ее работы. Например, в них могут инициализироваться переменные, открываться нужные файлы и т.д. В нашем примере в разделе инициализации инициализируется переменная Count.

## Раздел деинициализации

В этом разделе размещаются исполняемые операторы, содержащие некоторый фрагмент программы который выполняется в момент окончания работы программы. Иницилирующая и завершающая части модуля, если они есть, то при старте программы выполняются иницилирующие части всех модулей в порядке их перечисления в заголовке программы (в предложении **uses** файла проекта). Завершающие части при окончании работы программы выполняются в обратном порядке. В выше приведенном примере раздел деинициализации отсутствует.

## Файл проекта

Файл в котором находится основная программа называется файлом проекта и имеет расширение `dpr`. Кроме того что файл проекта содержит основную программу он играет очень важную роль в работе интегрированной среды Delphi. Этот файл содержит список всех файлов используемых проектом.

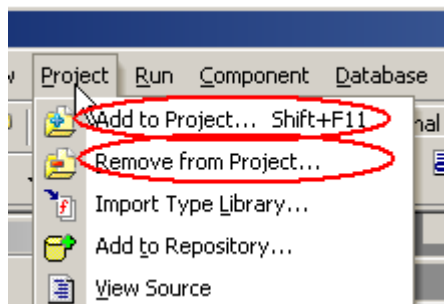
Список файлов находится в разделе **uses** в виде строк следующего вида

<имя модуля> **in** <имя файла>

Например модуль Tables включенный в проект будет выглядеть следующим образом

```
program TableRec;  
{$APPTYPE CONSOLE}  
uses  
  Tables in 'Tables.pas';
```

Непосредственно изменять файл проекта не рекомендуется. Удобнее делать это используя команды меню Project «Add to Project» и «Remove from Project» (рисунок ) которые соответственно добавляют файл в проект и удаляют файл из проекта.



 на

Рисунок 13 – Команды меню для работы с файлом проекта

Просмотреть все файлы входящие в проект можно нажав кнопку View Unit панели инструментов.

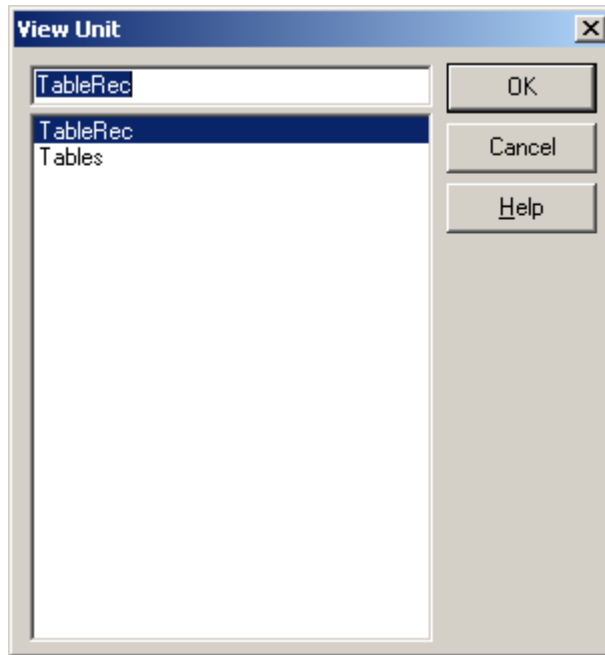


Рисунок 14 – Окно списка модулей включенных в проект

Преимуществом включения файла модуля в проект является то, что он контролируется интегрированной средой разработки и своевременно обновляется, т. е.

синхронизируется файл с исходным кодом модуля (pas) и откомпилированный файл модуля (dcu). При поиске файла модуля система сначала просматривает именно файлы включенные в проект, и, в случае неудачи, ищет файл согласно выше приведенному [алгоритму](#). При создании нового модуля он автоматически добавляется в проект.

## Стандартные модули

В поставку Delphi входит большое количество готовых модулей (несколько сотен).

Рассмотрим назначение некоторых из них.

### **Модуль System**

Модуль System подключается автоматически даже если список модулей пуст. Модуль System содержит основные подпрограммы переменные и константы необходимые для работы программы. Например такие подпрограммы как `writeln`, `readln`, `exp`, `sin`, `cos` и многие другие. Модуль System также осуществляет связь программы с операционной системой.

Таблица 17 – Математические функции модуля System

<i>Функция</i>	<i>Описание</i>
<b>function</b> Abs(X);	Возвращает модуль аргумента
<b>function</b> ArcTan(const X: Extended): Extended;	Возвращает арктангенс аргумента
<b>function</b> Cos(const X: Extended): Extended;	Возвращает косинус аргумента
<b>function</b> Sin(const X: Extended): Extended;	Возвращает синус аргумента
<b>function</b> Exp(const X: Extended): Extended;	Возвращает результат возведения числа e в степень аргумента
<b>function</b> Frac(const X: Extended): Extended;	Возвращает дробную часть аргумента
<b>function</b> Int(const X: Extended): Extended;	Возвращает целую часть аргумента
<b>function</b> Ln(const X: Extended): Extended;	Возвращает натуральный логарифм аргумента
<b>function</b> Pi;	Возвращает значение числа Пи
<b>function</b> Sqr(const X: Extended): Extended;	Возвращает квадрат аргумента
<b>function</b> Sqrt(const X: Extended): Extended;	Возвращает квадратный корень аргумента

Таблица 18 – Подпрограммы для работы со строками

<i>Функция</i>	<i>Описание</i>
<b>function</b> Copy (Source: string; Index, Count: Integer):string;	Копирует из строки Source Count символов, начиная с символа с номером Index.
<b>procedure</b> Delete (var Source: string; Index, Count: Integer);	Удаляет Count символов из строки Source, начиная с символа с номером индекс.
<b>procedure</b> Insert (SubSt: string; var St: string; Index: Integer);	Вставляет подстроку SubSt в строку St, начиная с символа с номером Index.
<b>function</b> Length (S:string):integer;	Возвращает текущую длину строки.

<b>function</b> Pos (SubSt, St:string):integer;	Отыскивает в строке St первое вхождение подстроки SubSt и возвращает номер позиции с которой она начинается. Если подстрока не найдена возвращается ноль.
<b>function</b> Concat(s1 [, s2,..., sn]: string): string;	Выполняет объединение нескольких строк в одну
<b>procedure</b> Str(X [:Width [:Decimals ]];var S);	Преобразует численное значение аргумента в его строковое представление
<b>procedure</b> Val(S; var V; var Code: Integer);	Преобразует строковый аргумент S в его численное представление V

Таблица 19 – Подпрограммы для работы с переменными порядковых типов

<i>Функция</i>	<i>Описание</i>
<b>procedure</b> Dec(var X[ ; N: Longint]);	Уменьшает значение X на N или единицу если не указан второй аргумент.
<b>procedure</b> Inc(var X [ ; N: Longint ] );	Увеличивает значение X на N или единицу если не указан второй аргумент
<b>function</b> Odd(X: Longint): Boolean;	Проверяет является ли аргумент нечетным числом
<b>function</b> Pred(X);	Возвращает предыдущий элемент из списка значений порядкового типа
<b>function</b> Succ(X);	Возвращает последующий элемент из списка значений порядкового типа

Таблица 20 – Функции преобразования типов

<i>Функция</i>	<i>Описание</i>
<b>function</b> Chr(X: Byte): Char;	Возвращает символ с заданным порядковым номером
<b>function</b> High(X);	Возвращает верхний предел диапазона значений аргумента
<b>function</b> Low(X);	Возвращает нижний предел диапазона значений аргумента
<b>function</b> Ord(X);	Возвращает порядковый номер значения порядкового типа
<b>function</b> Round(X: Extended): Int64;	Округляет значение вещественного типа к ближайшему целому
<b>function</b> Trunc(X: Extended): Int64;	Преобразует значение вещественного типа к целому путем отсечения дробной части

В модуле System также находятся функции для работы с файлами и динамической памятью, а также некоторые другие подпрограммы.



## Модуль Math

В поставку Delphi входит модуль Math в который, как можно догадаться по названию, входят часто используемые математические функции. Преимуществом использования функций из этого модуля является то, что они очень быстро работают, так как написаны на языке ассемблера. Ниже приведена таблица некоторых функций модуля Math.

Таблица 21 – Тригонометрические и гиперболические функции

<i>Функция</i>	<i>Описание</i>
<b>Тригонометрические и обратные тригонометрические функции</b>	
<b>function</b> CoTan( <b>const</b> X: Extended): Extended;	Котангенс
<b>function</b> Tan( <b>const</b> X: Extended): Extended;	Тангенс
<b>procedure</b> SinCos( <b>const</b> Theta: Extended; <b>var</b> Sin, Cos: Extended);	Вычисление синуса и косинуса угла, работает в два раза быстрее, чем если вычислять отдельно cos и sin.
<b>function</b> ArcSin( <b>const</b> X: Extended): Extended;	Арксинус
<b>function</b> ArcCos( <b>const</b> X: Extended): Extended;	Арккосинус
<b>function</b> ArcTan2( <b>const</b> X: Extended): Extended;	Арктангенс с учетом квадранта(функция ArcTan, не учитывающая квадрант, находится в модуле System)
<b>Гиперболические функции</b>	
<b>function</b> ArcCosh( <b>const</b> X: Extended): Extended;	Гиперболический арккосинус

<b>function</b> ArcSinh( <b>const</b> X: Extended): Extended;	Гиперболический арксинус
<b>function</b> ArcTahn( <b>const</b> X: Extended): Extended;	Гиперболический арктангенс
<b>function</b> Sinh( <b>const</b> X: Extended): Extended;	Гиперболический синус
<b>function</b> Tanh( <b>const</b> X: Extended): Extended;	Гиперболический тангенс
<b>function</b> Cosh( <b>const</b> X: Extended): Extended;	Гиперболический косинус

### Функции преобразования

<b>function</b> CycleToRad( <b>const</b> Cycles: Extended): Extended;	Преобразование циклов в радианы
<b>function</b> DegToRad ( <b>const</b> Degrees: Extended): Extended;	Преобразование градусов в радианы
<b>function</b> GradToRad ( <b>const</b> Grads: Extended): Extended;	Преобразование градусов в радианы
<b>function</b> RadToCycle ( <b>const</b> Radians: Extended): Extended;	Преобразование радианов в циклы
<b>function</b> RadToDeg ( <b>const</b> Radians: Extended): Extended;	Преобразование радианов в градусы
<b>function</b> RadToGrad ( <b>const</b> Radians: Extended): Extended;	Преобразование радианов в градусы

Таблица 22 – Арифметические функции модуля Math  
*Функция*

	<i>Описание</i>
<b>function</b> Cell ( <b>const</b> X: Extended): Integer;	Округление вверх
<b>function</b> Floor( <b>const</b> X: Extended): Integer;	Округление вниз

<b>function</b> IntPower( <b>const</b> Base: Extended; <b>const</b> Exponent: Integer): Extended;	Возведение числа в целую степень. Работает быстрее, чем функция Power.
<b>function</b> Ldexp ( <b>const</b> X: Extended; <b>const</b> P: Integer): Extended;	Умножение X на 2 в степени P
<b>function</b> LnXP1 ( <b>const</b> X: Extended): Extended;	Вычисление натурального логарифма X+1. Рекомендуется для X, близких к нулю
<b>function</b> LogN ( <b>const</b> Base, X: Extended): Extended;	Вычисление логарифма X по основанию N
<b>function</b> Log10 ( <b>const</b> X: Extended): Extended;	Вычисление десятичного логарифма X
<b>function</b> Log2 ( <b>const</b> X: Extended): Extended;	Вычисление двоичного логарифма X
<b>function</b> Power ( <b>const</b> Base, Exponent: Extended): Extended;	Возведение числа в степень. Работает медленнее IntPower.

Кроме того модуль Math содержит множество подпрограмм для статистической обработки данных, а также некоторые функции используемые в финансовом анализе. Более подробную информацию о модуле Math можно получить воспользовавшись справочной системой.

## **Модуль SysUtils**

Модуль SysUtils содержит большое количество вспомогательных подпрограмм таких как, например, подпрограмм преобразования типов (см. Лабораторную работу №2).

Ввиду частого применения подпрограмм из этого модуля, модуль SysUtils автоматически добавляется в шаблон приложения.

Некоторые из функций этого модуля были рассмотрены ранее, другие функции будут рассматриваться в следующих лабораторных работах. Кроме модуля SysUtils часто используются еще два модуля. Это модули Windows и Messages которые обеспечивают вызов функций Windows API из программ Object Pascal.

## Задания к лабораторной работе

1. Перепишите задание 3 из лабораторной работы №7 с использованием модуля.
2. Создайте модуль для работы с массивами. В состав модуля должны войти подпрограммы для ввода и вывода массивов, вставки и удаления элементов, несколько видов сортировки. Напишите тестовый проект для демонстрации возможностей модуля.
3. Перепишите программу из листинга 7 лабораторной работы №6 с использованием модуля. Интерфейсная часть модуля должна быть идентична модулю созданному в первом задании.

## Вопросы к лабораторной работе

1. Сколько и какие разделы содержит модуль Object Pascal?
2. Каково назначение каждого раздела модуля?
3. Как подключить модуль к программе?
4. В чем основное различие между разделом интерфейса и разделом реализации модуля?
5. В каком порядке выполняются разделы инициализации модулей?
6. Сколько предложений uses может содержать модуль?
7. Каково назначение файла проекта?
8. Как добавить модуль в файл проекта?
9. Как создать файл модуля?
10. В чем отличие между файлом модуля с исходным текстом и откомпилированным файлом модуля?
11. Каков алгоритм поиска файла модуля если он включен в файл проекта, если не включен?

## Справочные таблицы

Таблица 1 – Математические функции модуля System .....	95
Таблица 2 – Подпрограммы для работы со строками .....	95
Таблица 3 – Подпрограммы для работы с переменными порядковых типов .....	96
Таблица 4 – Функции преобразования типов .....	96
Таблица 5 – Тригонометрические и гиперболические функции .....	97
Таблица 6 – Арифметические функции модуля Math .....	98

# Лабораторная работа №8

## Динамическая память и указатели

### Типы с управляемым временем жизни.

## Введение

В данной лабораторной работе рассматриваются указатели, а также их использование в Object Pascal. Рассмотрены типизированные и нетипизированные указатели, функции для работы с динамической памятью. Также рассмотрены некоторые типы с управляемым сроком жизни, такие как длинные строки и динамические массивы.

## Динамическая память

При загрузке в память программа располагается в ней следующим образом

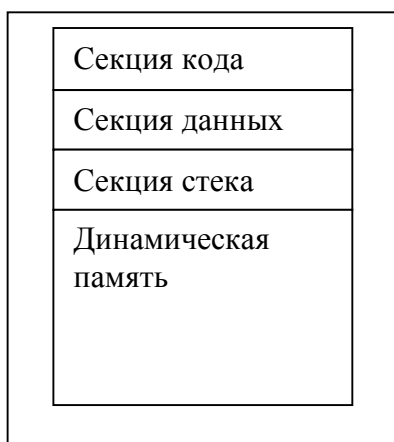


Рисунок 15 – Вид программы в ОЗУ

Рассмотрим подробнее назначение каждой части программы. Секция кода – это та часть программы где хранится исполняемый код модулей, а также основной программы. Далее располагаются глобальные переменные и типизированные константы. После данных следует область стека. **Стек** – это специальным образом организованная область памяти для временного хранения данных. Обычно стек используется для хранения фактических параметров передаваемых в подпрограммы, а также локальных переменных и констант. Под динамическую память отводится вся оставшаяся память компьютера<sup>4</sup>.

Таким образом динамическая память – это оперативная память ПК, предоставляемая программе при ее работе. В отличие от статического размещения данных, осуществляемого компилятором **Object Pascal** в процессе компиляции программы, при динамическом размещении заранее не известны ни тип, ни количество размещаемых данных.

Учитывая что при работе программа может использовать любые ячейки памяти, то необходимо вести учет занятых или учет свободных ячеек памяти. Иначе возможна ситуация когда различные части программы будут использовать одну и ту же область памяти, что может привести к порче данных и краху приложения. Эту функцию выполняет специальная часть программы называемая **менеджером памяти**. Менеджер памяти находится в модуле System. Доступ к менеджеру памяти осуществляется посредством специальных подпрограмм модуля System которые будут приведены ниже.

<sup>4</sup> Windows каждому приложению выделяет 2Гб адресного пространства, поэтому фактически на динамическую память остается 2Гб за вычетом размера программы.

## Указатели

Оперативная память ПК представляет собой совокупность ячеек для хранения информации - байтов, каждый из которых имеет собственный номер. Эти номера называются адресами, они позволяют обращаться к любому байту памяти. Минимально адресуемая единица памяти в ПК это байт. **Object Pascal** предоставляет в распоряжение программиста гибкое средство управления динамической памятью – указатели.

**Указатель** – это переменная, которая в качестве своего значения содержит адрес ячейки памяти. С помощью указателей можно размещать в динамической памяти любой из известных в **Object Pascal** типов данных. Лишь некоторые из них (Byte, Char, ShortInt, Boolean) занимают во внутреннем представлении один байт, остальные – несколько смежных. Поэтому на самом деле указатель адресует лишь первый байт данных.

### Типизированные указатели

Указатель может связываться с некоторым типом данных. Такие указатели называют типизированными. Для объявления типизированного указателя используется значок ^, который помещается перед соответствующим типом, например:

```
var p1: ^Integer; //указатель на целочисленную переменную
    p2: ^Real;    //указатель на вещественное число

//указатель на переменную типа запись
type PersonPointer = ^PersonRecord;
    PersonRecord = record
        Name: String;
        Job : String;
        Next : PersonPointer
    end;
```

Как уже отмечалось, в **Object Pascal** действует принцип, в соответствии с которым перед использованием какого-либо идентификатора он должен быть описан. Исключение сделано только для указателей, которые могут ссылаться на еще не объявленный тип данных.

### Нетипизированные указатели

Указатели не связанные с каким-либо конкретным типом данных называются нетипизированными. В **Object Pascal** для этого служит стандартный тип Pointer. Поскольку нетипизированные указатели не связаны с конкретным типом, с их помощью удобно динамически размещать данные, структура и тип которых меняются в ходе работы программы.

```
var p: pointer; //нетипизированный указатель
```

### Операции с указателями

#### Операции присваивания и сравнения

Поскольку в **Object Pascal** реализована строгая типизация данных, то можно присваивать значения только указателей, связанных с одним и тем же типом данных. Если, например,

```
var p11,p12: ^Integer;
    pR: ^Real;
    p : Pointer;
```

то присваивание

```
p11:= p12
```

допустимо, в то время как

```
p11:= pR
```

запрещено, поскольку p11 и pR указывают на разные типы данных. Это ограничение, однако, не распространяется на нетипизированные указатели, поэтому можно записать следующее присваивание

```
p11:= p  
pR:=p;
```

В **Object Pascal** определена только одна логическая операция над указателями – это операция определения эквивалентности. Два указателя эквивалентны, если содержат один и тот же адрес, например

```
var a, b:integer; //целочисленная переменная  
    p, p1:^integer; //указатель на целочисленную переменную  
begin  
    ...  
    p:=@a; {вычисляем адрес переменной a и помещаем его в  
переменную p}  
    p1:=@b;  
    if p=p1 then writeln('Equivalent')  
        else writeln('not Equivalent'); {на экран будет выведено not  
Equivalent, т.к. переменные a и b имеют различные адреса}  
    ...  
end.
```

Кроме того определена одна константа – пустой указатель **nil**. Эта константа служит для определения указателей которым еще не присвоен адрес какой-либо переменной с целью исключения обращения программы по несуществующему или неправильному адресу.

### Определение адреса переменной

Для определения адреса переменной используется оператор @, например

```
var a:integer; //целочисленная переменная  
    p:^integer; //указатель на целочисленную переменную  
begin  
    ...  
    p:=@a; {вычисляем адрес переменной a и помещаем его в  
переменную p}  
    ...  
end.
```

Обратите внимание на то, что тип переменной адрес которой вычисляется и тип

указателя должны совпадать. Исключение составляет нетипизированный указатель ему

можно присвоить адрес любой переменной.

### Операция разадресации указателя

Чтобы получить доступ к данным адрес которых содержит указатель необходимо применить к указателю операцию разадресации. Сделать это можно с помощью оператора ^, например

```
var a:integer; //целочисленная переменная  
    p:^integer; //указатель на целочисленную переменную  
begin  
    ...  
    p:=@a; {вычисляем адрес переменной a и помещаем его в
```

```

переменную p}
p^:=1;{в ячейку памяти адрес которой содержится в указателе p
записываем единицу}
writeln(a);{на экран будет выведена единица}
...
end.

```

Таким образом, значение, на которое указывает указатель, т.е. собственно данные, размещенные в памяти, обозначаются значком ^, который ставится сразу за указателем.

Если за указателем нет значка ^, то имеется в виду адрес, по которому размещены данные.

## Арифметические операции с указателями

В **Object Pascal** определены две арифметические операции над указателями. Это увеличение и уменьшение указателя. Производятся эти операции с помощью функций **inc** и **dec** модуля **System**. Особенностью работы этих функций с **типизированными указателями** является то, что увеличение или уменьшение указателя происходит на величину кратную размеру типа на который ссылается указателя, например

Листинг 46 – Арифметические операции с указателями

```

program IncDecPointer;
{$APPTYPE CONSOLE}
const n=5;
var a:array [1..n] of Integer=(1, 2, 3, 4, 5);
    e1:^Integer;
    i:Integer;

begin
  e1:=@a; //присваиваем указателю адрес первого элемента массива

  for i:=1 to n do
    begin
      writeln(e1^); //вывод текущего элемента
      inc(e1); //увеличение указателя
    end;
  readln;
end.

```

В программе приведенной выше указатель при применении функции **inc** увеличивается на 4 байта, размер типа **integer**. Если написать

```
inc(e1, 2);
```

то указатель изменится на 8 байт, т.к. две переменные типа **integer** занимают 8 байт.

При применении функций **inc** или **dec** к **нетипизированным указателям** увеличение или уменьшение указателя происходит на один байт.

## Выделение и освобождение динамической памяти

Вся динамическая память в **Object Pascal** рассматривается как сплошной массив байтов, который называется кучей.

Для работы с **типизированными указателями** применяется пара функций **new** и **dispose**.



Для выделения памяти под динамически размещаемую переменную необходимо воспользоваться процедурой **new**. Параметром обращения к этой процедуре является типизированный указатель. В результате обращения указатель приобретает значение, соответствующее адресу, начиная с которого можно разместить данные.

Динамическую память можно не только забирать из кучи, но и возвращать обратно (более того это просто необходимо делать иначе весь объем динамической памяти может быть исчерпан). Для этого используется процедура **dispose**, например:

Листинг 47 – Работа с типизированными указателями

```
program DinMem;
{$APPTYPE CONSOLE}
var pInt:^Integer;
begin
  new(pInt); //создаем динамическую переменную
  pInt^:=15; //записываем значение в переменную
  writeln(pInt^); //печать
  dispose(pInt); {динамическая переменная больше не нужна удаляем ee}
  readln;
end.
```

Динамически размещенные данные можно использовать в любом месте программы, где это допустимо для констант и переменных соответствующего типа.

Необходимо отметить, что процедура **dispose**(pInt) не изменяет значения указателя pInt, а лишь возвращает в кучу память, ранее связанную с этим указателем. Однако повторное применение процедуры к свободному указателю приведет к возникновению ошибки периода исполнения. Для решения этой проблемы освободившемуся указателю можно присвоить константу **nil**. Поэтому более корректным будет следующий код,

Листинг 48 – Использование константы nil

```
program DinMem;
{$APPTYPE CONSOLE}
var pInt:^Integer;
begin
  new(pInt); //создаем динамическую переменную
  pInt^:=15; //записываем значение в переменную
  writeln(pInt^); //печать
  dispose(pInt); {динамическая переменная больше не нужна удаляем ee}
  pInt=nil; //присваиваем пустой указатель
  readln;
end.
```

В выше приведенном примере такое присваивание конечно не обязательно, но если указатель будет использоваться после вызова процедуры **dispose**, например, снова будет выделена память и т.п., то следует для повышения безопасности программы такое присваивание выполнять. Ниже приведенный пример показывает, как можно избежать краха программы, из-за неправильного использования указателей применяя константу **nil**.

Листинг 49

```
program DinMem2;
{$APPTYPE CONSOLE}
var pInt:^Integer;
begin
  new(pInt); //создаем динамическую переменную
  pInt^:=15; //записываем значение в переменную
  writeln(pInt^); //печать
  dispose(pInt); {динамическая переменная больше не нужна удаляем ee}
  pInt=nil; //присваиваем пустой указатель
```

```

... //какой-то код
dispose(pInt); //здесь может произойти ошибка
readln;
end.

```

Если в приведенном примере закомментировать строку `pInt=nil`, то произойдет ошибка времени исполнения. Использование указателей, которым не присвоено значение процедурой **new** или другим способом, также вызовет исключительную ситуацию и крах программы.

Для работы с **нетипизированными указателями** используется пара процедур **GetMem** и **FreeMem**. Процедура **GetMem** служит для резервирования памяти и имеет следующий синтаксис

```

procedure GetMem(P, Size);

```

где P – нетипизированный указатель;

Size – размер в байтах требуемой или освобождаемой части кучи.

Процедура **FreeMem** служит для освобождения памяти и имеет следующий синтаксис

```

procedure FreeMem(P, [Size]);

```

Использование процедур **GetMem/FreeMem**, как и вообще вся работа с динамической памятью, требует особой осторожности и тщательного соблюдения простого правила: освобождать нужно ровно столько памяти, сколько ее было зарезервировано, и именно с того адреса, с которого она была зарезервирована<sup>5</sup>.

Также следует отметить, что **не следует смешивать вызовы** функций `new/dispose` и `GetMem/FreeMem`, т.е. нельзя выделить память с помощью процедуры **new**, а освободить с помощью **FreeMem** т.к. это вызовет крах программы.

В качестве примера применения процедур **GetMem/FreeMem** рассмотрим программу копирования файла из лабораторной работы №6. В программе применяется буфер в виде статической переменной, напишем программу использующую в качестве буфера динамическую переменную.

Листинг 50 – Программа копирования файлов

```

program FileCopy;

{$APPTYPE CONSOLE}
const BufSize=1024; //размер буфера
var Buf:Pointer; //буфер
    F1, F2:file;
    FileName1,
    FileName2:string;
    BytesTransferred:Integer;
    Size:Integer;
begin
  writeln('vvedite filename');
  readln(FileName1);

  //открываем копируемый файл
  AssignFile(F1, FileName1);
  {$I-}
  Reset(F1, 1);
  if IOResult<>0 then halt(1);

```

<sup>5</sup> Начиная с Delphi 4 параметр Size в процедуре **FreeMem** игнорируется и оставлен для совместимости со старыми программами. Менеджер памяти Object Pascal автоматически запоминает размер выделенного блока памяти с помощью процедуры **GetMem** и использует это значение при вызове **FreeMem**.

```

//создаем файл копии
writeln('copy file');
readln(FileName2);
AssignFile(F2, FileName2);
Rewrite(F2, 1);
if IOResult<>0 then
begin
  CloseFile(F1);
  Halt(1);
end;
{$I+}

//выделяем память под буфер
GetMem(Buf, BufSize);

//копируем файл
while not EOF(F1) do
begin
  Size:=BufSize;
  BlockRead(F1, Buf^, Size, BytesTransferred);
  Size:=BytesTransferred;
  BlockWrite(F2, Buf^, Size);
end;

CloseFile(F1);
CloseFile(F2);
//освобождаем память отведенную под буфер
FreeMem(Buf);
end.

```

В таблице.1 приводится описание как уже рассмотренных подпрограмм **Object Pascal**, так и некоторых других, которые могут оказаться полезными при обращении к динамической памяти. Все подпрограммы работы с динамической памятью находятся в модуле System.

Таблица 23 – Подпрограммы Object Pascal для работы с памятью

<i>Подпрограмма</i>	<i>Описание</i>
<b>function</b> Addr(X): Pointer;	Возвращает адрес аргумента X. Аналогичный результат возвращает операция @,.
<b>procedure</b> Dispose (var P: Pointer);	Возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за типизированным указателем P
<b>procedure</b> FreeMem(var P: Pointer; Size: Integer);	Возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за нетипизированным указателем.
<b>procedure</b> GetMem(var P: Pointer; Size: Integer) ;	Резервирует за нетипизированным указателем фрагмент динамической памяти требуемого размера Size.
<b>procedure</b> New(var P: Pointer);	Резервирует фрагмент кучи для размещения переменной и помещает в типизированный указатель P адрес первого байта.
<b>function</b> SizeOf(X): Integer;	Возвращает длину в байтах внутреннего представления указанного объекта X.

## Типы с управляемым временем жизни

Некоторые типы **Object Pascal** по сути являются указателями, хотя программист никогда для переменных этих типов не выделяет и не освобождает память. Более того не используется операция разадресации указателей. Вся работа по выделению и освобождению памяти для переменных этих типов производится автоматически. Операция разадресации также производится автоматически. К этим типам относятся длинные строки (AnsiString), динамические массивы и интерфейсы. Общим для всех этих типов является то, что после того как программа перестает использовать переменную (например после выхода из подпрограммы) любого из этих типов, то память занятая переменной автоматически освобождается. Таким образом, в отличие от обычных динамических переменных, при работе с переменными с управляемым временем жизни нет необходимости (или почти нет) следить за своевременным освобождением памяти, что существенно повышает надежность программ.

### Длинные строки

В стандартном Паскале используются только короткие строки String[N]. В памяти такой строке выделяется N+1 байт, первый байт содержит текущую длину строки, а сами символы располагаются начиная со 2-го по счету байта. Поскольку для длины строки в этом случае отводится один байт, максимальная длина короткой строки не может превышать 255 символов. Для объявления короткой строки максимальной длины может использоваться стандартный тип ShortString (эквивалент String [255]).

Директива компилятора `{ $H }` определяет какой тип строки будет подразумеваться под типом String. По умолчанию `{ $H+ }` String – длинная строка. При установке `{ $H- }` под типом String будет подразумеваться тип ShortString.

В Windows используются только нуль-терминальные строки<sup>6</sup>, представляющие собой цепочки символов, ограниченные символом #0. Максимальная длина такой строки лимитируется только доступной памятью и может быть очень большой.

В 32-разрядных версиях Delphi введен новый тип AnsiString, сочетающий в себе удобства обоих типов. При работе с этим типом память выделяется по мере надобности (динамически) и ограничена имеющейся в распоряжении программы доступной памятью.

Механизм работы с памятью для длинных строк следующий. При объявлении длинной строки S: компилятор выделит для переменной 4 байта, достаточные для размещения указателя на область памяти где будут находиться данные строки и установит значение указателя в **nil**. При присваивании строке значения «Hello» программа (а не компилятор!) определит длину цепочки символов «Hello», обратится к ядру операционной системы с требованием выделить для нее участок памяти длиной 5+5=10 байт, поместит в переменную S указатель на выделенную область памяти. Далее программа скопирует в эту область памяти данные строки, т.е. «Hello», завершив ее терминальным нулем и 4-байтным счетчиком ссылок.

Такое размещение на этапе выполнения программы называется динамическим, в то время как размещение на этапе компиляции – статическим. При работе с длинными строками используется механизм подсчета ссылок. С его помощью реализуется «кэширование» памяти: при выполнении оператора

```
SS:=S;
```

память для размещения значения переменной SS не выделяется, в переменную SS помещается содержимое указателя S, а счетчик ссылок в связанной с ним памяти увеличивается на единицу. Таким образом, оба указателя будут ссылаться на одну и ту же

<sup>6</sup> На самом деле еще используются широкие строки, а также OLE строки.

область памяти, счетчик ссылок которой будет содержать значение 2. При выполнении оператора

```
SS:= S+' World';
```

счетчик ссылок уменьшается на единицу, выделяется новая область памяти длиной  $11 + 5 = 16$  байт, указатель на эту область помещается в SS, а в саму память переписывается цепочка символов «Hello World», терминальный ноль и содержащий единицу счетчик ссылок. Теперь переменные S и SS будут ссылаться на разные участки памяти, счетчики ссылок которых будут содержать по единице. Выделенная для размещения строки String область памяти освобождается, если ее счетчик ссылок стал равен нулю.

## Динамические массивы

Начиная с Delphi 4 появилась возможность наряду с обычными, статическими массивами, использовать динамические массивы, в которых не определено число элементов на этапе компиляции.

Пример объявления динамического массива

```
var a: array of integer;
```

Фактически динамические массивы, как и длинные строки, являются указателями, но в отличие от длинных строк память для динамических массивов необходимо выделять самостоятельно используя процедуру SetLength. Перед использованием динамического массива следует установить его длину с помощью процедуры SetLength имеющую следующий синтаксис

```
procedure SetLength(var S; NewLength: Integer)
```

где S – динамический массив;

NewLength – новая длина массива (число элементов).

После использования массив можно удалить из памяти с помощью функции Finalize(S) или NewLength(S, 0). Следует помнить, что динамические массивы всегда начинаются с нулевого индекса. Верхнюю границу динамического массива можно определить с помощью функции High(S).

Возможно также создание динамических многомерных массивов, например.

```
var a:array of array of Double; //двумерный массив
```

выделить память под многомерный массив можно различными способами например с помощью процедуры SetLength

```
SetLength(a, 10, 2); //массив 10x2 элемента
```

возможно выделение памяти для каждой колонки, ниже приведен пример создания двумерного массива с различным количеством колонок в строках.

```
const n=10;
var i:integer;
    a:array of single;
begin
  SetLength(a, n);
  for i:=1 to n+1 do SetLength(a[i-1], i)
end.
```

В данном примере в первой строке будет одна колонка, во второй две в n-й n+1 колонка. Для работы с подобными массивами удобно использовать функцию High для определения верхней границы массива, например заполним созданный выше массив единицами.

```
const n=10;
var i, j:integer;
```

```

    a:array of Single;
begin
    //создание массива
    SetLength(a, n);
    for i:=1 to n+1 do SetLength(a[i-1], i);
    //заполняем массив единицами
    for i:=0 to n do
        for j:=0 to High(a[i]) do
            a[i, j]:=1
        end;
    end.

```

Следует отметить, что при изменении размеров массива значения элементов массива не теряются, т.е. если уменьшить размер массива со 100 элементов до 50, то первые 50 элементов останутся неизменными. Но следует помнить, что при увеличении массива элементы будут добавлять в конец массива и их содержимое не предсказуемо.

Динамические массивы, в отличие от длинных строк не используют технологию подсчета ссылок, поэтому, для того чтобы выполнить операцию присвоения (получить копию) необходимо использовать стандартную функцию Copy имеющую следующий синтаксис.

**function Copy(S; Index, Count: Integer): array;**

где S – копируемый массив;

Index – индекс начиная с которого необходимо осуществить копирование;

Count – число копируемых элементов.

```

var a, b:array of Integer;
begin
    SetLength(a, 5);
    b:=a;
    a[0]:=5;
    b[0]:=10;
end.

```

После выполнения выше приведенного фрагмента a[0] примет значение 10, т.к. a и b указывают на одну и ту же область памяти.

```

var a, b:array of Integer;
begin
    SetLength(a, 5);
    b:=Copy(a);
    a[0]:=5;
    b[0]:=10;
end.

```

После выполнения последнего примера a[0] примет значение 5, т.к. a и b указывают на различные ячейки памяти.

Обратите внимание на то, что динамический массив является типом с управляемым временем жизни и, следовательно, если динамический массив был объявлен, например, внутри подпрограммы, то при выходе из нее память занятая динамическим массивом будет освобождена.

Все остальные операции определенные для статических массивов выполняются также и для динамических массивов.

## Задания к лабораторной работе

1. Наберите и изучите алгоритмы программ приведенных в лабораторной работе.

2. Напишите программу, производящую вычисление выражения  $x^3 + \sin(x) + a/b$ , все переменные должны быть динамическими.
3. Напишите программу создающую двумерный динамический массив содержащий 10 строк. Четные строки массива должны содержать по 5 столбцов, а нечетные по 3. Напишите код позволяющий осуществлять ввод и вывод данных в такой массив.
4. Перепишите программу из лабораторной работы №4 задание 4 с использованием динамических массивов. Размерность матриц должна задаваться с клавиатуры.
5. Перепишите программу из лабораторной работы №4 задание 7 с использованием динамических массивов. Число студентов вводимых в список должно задаваться с клавиатуры.

## Вопросы к лабораторной работе

1. Где располагается динамическая память программы?
2. Что такое указатель?
3. Что такое типизированный указатель и как он объявляется в Object Pascal?
4. Что такое нетипизированный указатель и как он объявляется в Object Pascal?
5. Какие операции допустимы с указателями?
6. С помощью какого оператора можно определить адрес переменной?
7. Что такое операция разадресации указателя?
8. Какие арифметические операции с указателями допустимы в Object Pascal.?
9. Какие особенности выполнения арифметических операций с типизированными указателями существуют в Object Pascal?
10. Для чего нужен менеджер памяти?
11. Какие функции выделения (освобождения) динамической памяти применяются при работе с типизированными указателями?
12. Какие функции выделения (освобождения) динамической памяти применяются при работе с нетипизированными указателями?
13. Можно ли создать динамическую переменную типа `integer` с помощью `GetMem`?
14. Для чего используется константа `nil`?
15. Что такое тип с управляемым временем жизни? В чем отличие такого типа от обычного?
16. Какие типы относятся к типам с управляемым временем жизни?
17. В чем заключается механизм подсчета ссылок использующийся при работе с длинными строками?
18. Как создать копию динамического массива, длинной строки?

## Справочные таблицы

Таблица 1 – Подпрограммы Object Pascal для работы с памятью .....107

## Лабораторная работа №9

### Динамические структуры данных. Односвязные списки.

#### Введение

В данной лабораторной работе рассмотрены односвязные списки. Дана структура для описания узлов односвязных списков. Рассмотрены основные алгоритмы работы со списками, такие как вставка и удаление узлов, поиск узлов, сортировка и др. Рассмотрен пример создания модуля для работы с односвязными списками. Рассмотрены некоторые примеры применения односвязных списков.

#### Динамические структуры данных

Динамическими структурами данных называют структуры данных размер и количество элементов которых может изменяться во время работы программы. Существует большое количество динамических структур данных. Примером таких структур могут служить списки и деревья. Рассмотрим одну из наиболее распространенных структур данных – списки.

#### Односвязные списки

Односвязный список представляет собой цепочку элементов, называемых узлами. При этом каждый элемент содержит указатель, указывающий на следующий элемент в этом списке.



Рисунок 16 – Узел односвязного списка

Для получения доступа ко всем элементам списка достаточно знать первый узел списка, от которого, путем переходов по ссылкам можно обойти все остальные узлы.

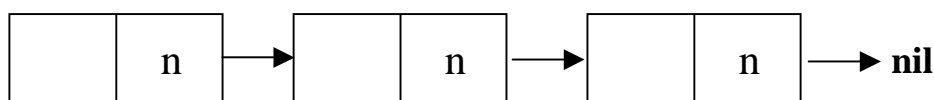


Рисунок 17 – Односвязный список

В отличие от массива узлы списка могут располагаться в различных участках памяти, а их порядок определяется ссылками. Таким образом размер списка ограничен лишь свободным объемом памяти. Недостатками списка является большое количество обращений к памяти при переборе элементов списка, как следствие, меньшее быстродействие при чтении списка. Достоинством списка является простота и высокая скорость вставки или удаления элемента из списка.

Конец списка обычно помечается пустым указателем `nil` или ссылка последнего узла указывает на первый узел, в этом случае список называется кольцевым.



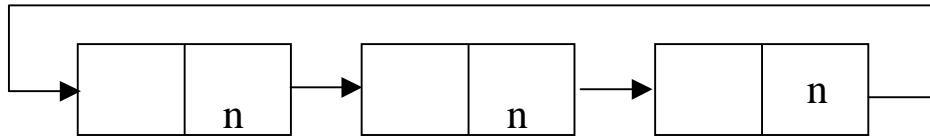


Рисунок 18 – Кольцевой односвязный список

## Узлы связного списка

Описание узлов односвязного списка приведено ниже

```
//Описание структуры узла
type PNode=^TNode; //указатель на узел
TNode=record
  next:PNode; //указатель на следующий узел
  data:pointer; //указатель на данные узла
end;
```

Узел может либо непосредственно содержать данные, либо содержать указатель на данные.

Для перехода по ссылке можно использовать примерно следующий код

```
var NextNode, CurrentNode:PNode;
begin
  ...
  NextNode:=CurrentNode^.next;
```

## Создание односвязного списка

В самом простом случае первый узел в связном списке описывает весь список. Первый узел называют головой списка.

```
var HeadNode:PNode;
```

Если **HeadNode** содержит **nil** значит списка еще нет. Таким образом список создается следующим образом

```
var HeadNode:Pnode=nil;
```

## Вставка и удаление элементов в односвязном списке

Для односвязного списка существует только один вариант вставки нового узла в список – вставкам после указанного узла списка. Пошаговый алгоритм вставки нового элемента в список приведен на рисунке 4

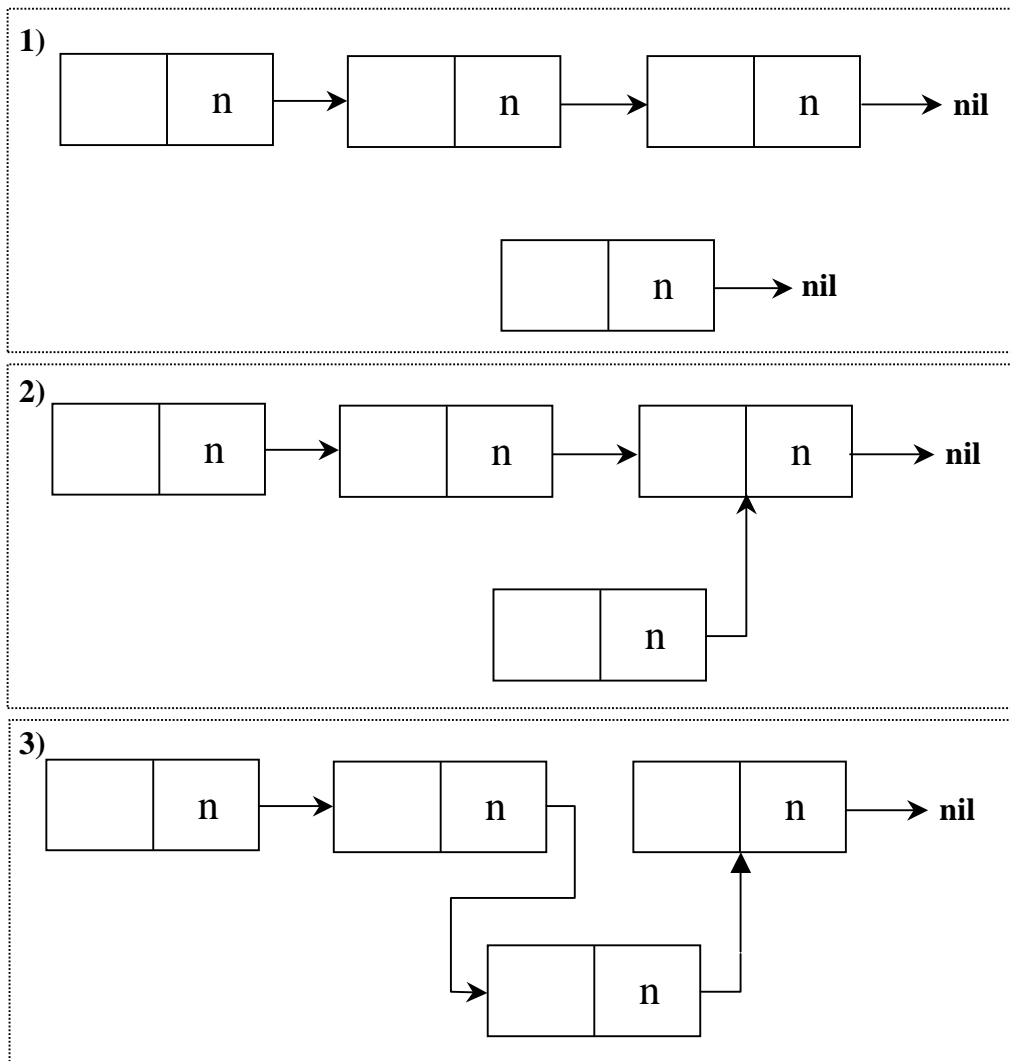


Рисунок 19 – Вставка нового узла в односвязный список

Подпрограмма реализующая данный алгоритм может выглядеть следующим образом.

```

function InsertBefore(var ANode:PNode; InsNode:PNode):PNode;
begin
  //если список не содержит узлов
  if ANode=nil then ANode:=InsNode
  else //если вставляемый узел не пустой
    if InsNode<>nil then
      begin
        InsNode^.next:=ANode^.next;
        ANode^.next:=InsNode;
      end;{if InsNode}
    Result:=InsNode;
  end;{InsertBefore}

```

Обратите что существует особый случай – это вставка перед первым узлом списка. В этом случае первым элементом в списке становится вставляемый элемент. Так как односвязный список задается своим первым элементом, то код для этого случая надо писать отдельно, например

```

var MyList:PNode=nil; //список
    InsNode:PNode
begin
  ...
  MyList:=InsertBefore(InsNode, MyList);
  ...
end.

```

Пошаговый алгоритм удаления узла из односвязного списка приведен на рисунке 5

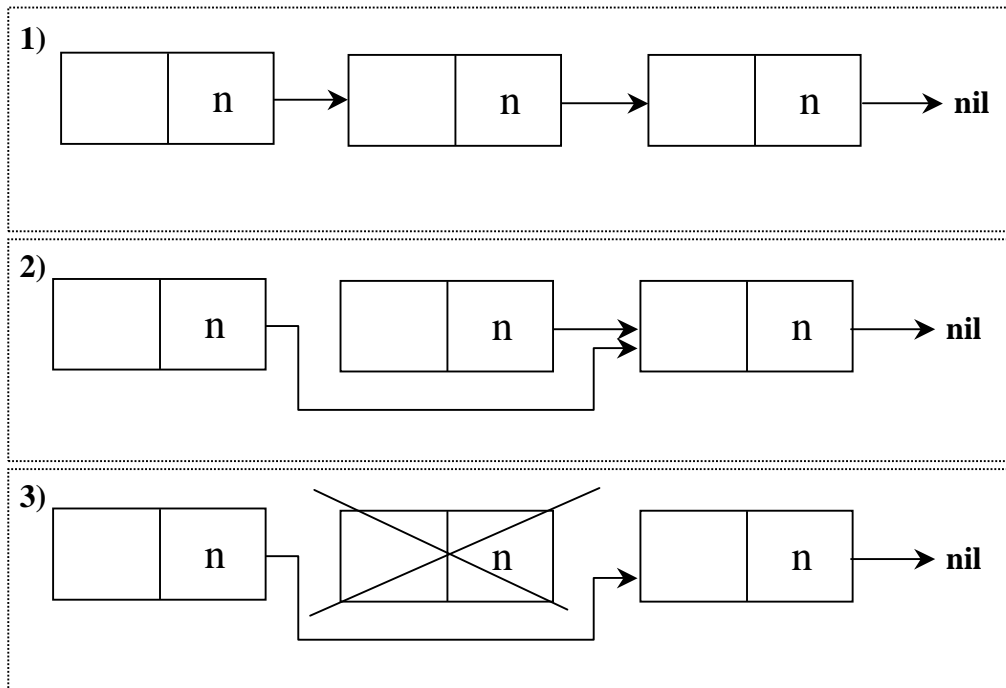


Рисунок 20 – Удаление узла из односвязного списка

Код удаления узла из списка может выглядеть следующим образом

```
function DeleteBefore(ANode:PNode):PNode;
begin
  if ANode<>nil then
    if ANode^.next<>nil then //если удаляемый узел существует
      begin
        Result:=ANode^.next;
        ANode^.next:=ANode^.next^.next;
        Result^.next:=nil;
      end{if ANode^.next}
    else Result:=nil
  else Result:=nil;
end;{DeleteBefore}
```

Здесь также существует особый случай, если удаляется первый элемент, он должен обрабатываться отдельно (не существует узел перед которым стоит первый узел), например так

```
var MyList:PNode=nil; //список
    Node:PNode
begin
  ...
  Node:=MyList;
  MyList:=MyList^.next;
  Dispose(Node);
  ...
end.
```

### Прохождение односвязного списка

Прохождение односвязного списка реализуется с помощью очень простого алгоритма:

- вводится текущий узел;
- текущему узлу присваивается первый узел списка;

- далее производится переход от узла к узлу с помощью указателей next. При этом для каждого узла может вызываться некоторая подпрограмма обрабатывающая данные содержащиеся в узле;
- процесс останавливается при достижении указателя **nil**.

Код реализующий данный алгоритм может выглядеть следующим образом

```

procedure ProcessList(AHead:PNode; ANodeProc:TProcessNode);
begin
  //обход списка начиная с первого узла
  while AHead<>nil do
    begin
      ANodeProc(AHead^.data); //процедура обработки данных узла
      AHead:=Next(AHead);
    end;{while}
  end;{ProcessList}

```

### Поиск предшествующего узла в односвязном списке

Часто необходимо найти в списке узел, стоящий перед заданным узлом. Алгоритм поиска такого узла прост, необходимо осуществлять проход списка, начиная с первого элемента до тех пор, пока не будет найден узел указатель next которого, указывает на заданный узел. Пример реализации кода реализующего этот алгоритм приведен ниже.

```

function Prev(AHead, ANode:PNode):PNode;
begin
  Result:=nil;
  //поиск узла указатель next которого равен ANode
  while AHead<>nil do
    begin
      if AHead^.next=ANode then
        begin
          Result:=AHead;
          break;
        end;{if}
      AHead:=Next(AHead);
    end;{while}
  end;{Prev}

```

### Вставка перед заданным узлом

Для осуществления вставки элемента перед заданным узлом списка необходимо найти узел предшествующий заданному узлу списка, а затем выполнить вставку после этого узла. Обратите внимание, что здесь также существует особый случай, а именно вставка перед первым узлом.

### Сортировка вставками

Продолжим рассматривать алгоритмы сортировки. Следующий алгоритм сортировки называется алгоритм сортировки вставками. Алгоритм сортировки заключается в следующем:

1. создаем пустой список, который будет содержать отсортированный список;
2. поиск «минимального» узла в исходном списке;
3. найденный узел удаляется из исходного списка и вставляется в отсортированный список;

4. изымается первый элемент из исходного списка и вставляется в новый список так, чтобы не нарушить его правильный порядок элементов (т.е. так чтобы после вставки элемента массив остался отсортированным);
5. пункт 4 повторяется до тех пор, пока в исходном списке не останется элементов.

Код реализующий данный алгоритм приведен ниже

```

procedure InsertionSort(var AHead:PNode;
CompareProc:TCompareProc);
var TempList:PNode;
    CurNode, Cur, PrevNode:PNode;
begin
    {создаем список, содержащий минимальный элемент исходного
    списка}
    TempList:=SearchMin(AHead, CompareProc);
    //удаляем найденный минимальный элемент из исходного списка
    TempList:=DeleteNode(AHead, TempList);

    //обход исходного списка начиная с первого элемента
    while AHead<>nil do
        begin
            {удаляем текущий элемент из исходного списка и
            переходим к следующему элементу}
            Cur:=AHead;
            AHead:=AHead^.next;
            Cur^.next:=nil;

            CurNode:=TempList; //установка на начало списка
            PrevNode:=CurNode;

            while CurNode<>nil do //поиск места вставки
                begin

                    if CompareProc(CurNode^.data, Cur^.data)>=0 then
                        begin
                            {если вставляемый элемент "меньше" текущего, то
                            вставляем его}
                            InsertBefore(PrevNode, Cur);
                            break;
                        end{if}
                    else
                        begin
                            PrevNode:=CurNode; //запоминаем текущий узел
                            CurNode:=Next(CurNode); //переходим к следующему узлу
                        end{else}
                    end;{ while CurNode}

                    {вставка элемента в конец списка}
                    if CurNode=nil then InsertBefore(PrevNode, Cur);
                end;{while AHead}

                //указатель на отсортированный список
                AHead:=TempList;
            end;{InsertionSort}

```

### ***Примеры программ использующих односвязные списки***

В данном разделе приведены некоторые примеры использования односвязных списков. Для работы всех примеров необходимо подключить к проекту модуль Lists. В модуле Lists собраны подпрограммы для работы с односвязными списками.

## Демонстрационная программа для модуля Lists

В ниже приведенной программе приведен пример использования некоторых подпрограмм из модуля Lists.

Листинг 51 – Демонстрационная программа

```
program SList;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Lists in 'Lists.pas';

var pInt:^integer;
    Head:PNode=nil; //список
    CurNode, Node:PNode; //текущий узел
    i, elm:integer;
    SNode:PNode;

//вспомогательные подпрограммы

//печать данных узла на экран
procedure NodePrint(P:Pointer);
begin
  if p<>nil then write(Integer(P^), ' ');
end;

//удаление целочисленных динамических переменных
procedure DestroyData(P:Pointer);
begin
  if p<>nil then dispose(p);
end;

//сравнение двух целых чисел
function CompareProc(P1, P2:Pointer):ShortInt;
begin
  if Integer(P1^)>Integer(P2^) then Result:=1
  else
    if Integer(P1^)<Integer(P2^) then Result:=-1
    else Result:=0;
end;

//условие отбора элементов списка
function CompareProc2(P1, P2:Pointer):ShortInt;
begin
  if Integer(P1^)>Integer(P2^) then Result:=0
  else result:=-1;
end;

//сравнение элемента списка с заданным элементом
function SearchProc(P1, P2:Pointer):ShortInt;
begin
  if Integer(P1^)=Integer(P2^) then Result:=0
  else result:=-1;
end;

//основная программа
begin
  writeln('Demo single link list');
  //создаем список содержащий 20 узлов
```

```

//каждый узел содержит целое число
randomize;
for i:=1 to 20 do
begin
  new(pInt); //динамическая переменная типа integer
  //присваиваем динамической переменной значение
  pInt:=random(100);
  Node:=NewNode(pInt); //создаем новый узел
  //вставляем узел в список
  if Head=nil then CurNode:=InsertBefore(Head, Node)
  else CurNode:=InsertBefore(CurNode, Node)
end;

//вывод содержимого списка на экран
writeln('Source List');
ProcessList(Head, NodePrint);
writeln;

//поиск элемента в списке
write('Input element to Search: ');
readln(elm);

if SearchNode(Head, @elm, SearchProc) <> nil then
  writeln('Element [' , elm, '] found')
else
  writeln('Element [' , elm, '] not found');

//удаление элемента из списка
write('Input element to delete: ');
readln(elm);

SNode:=SearchNode(Head, @elm, SearchProc);
if SNode <> nil then
begin
  writeln('Node [' , elm, '] deleted');
  DeleteNode(Head, SNode);
  ProcessList(Head, NodePrint);
  writeln;
end
else
  writeln('Element [' , elm, '] not found');

//вывод списка узлов значение которых больше заданной величины
write('Input value: ');
readln(elm);

SNode:=SearchNodes(Head, @elm, CompareProc2);
if SNode <> nil then
begin
  writeln('Search result');
  ProcessList(SNode, NodePrint);
  writeln;
  {уничтожение списка с результатами поиска,
  данные содержащиеся в списке не уничтожаются}
  DestroyList(SNode, nil);
end
else
  writeln('Elements not found');

//поиск максимального и минимального значений элементов

```

```

writeln('Min Value ', Integer(SearchMin(Head,
CompareProc)^.data^));
writeln('Max Value ', Integer(SearchMax(Head,
CompareProc)^.data^));

//сортировка списка
writeln('Sort List');
InsertionSort(Head, CompareProc);
ProcessList(Head, NodePrint);
writeln;

//уничтожение списка
DestroyList(Head, DestroyData);
writeln('Press Enter to exit');
readln;
end.

```

## Загрузка в список текстового файла

Следующая программа загружает в список содержимое текстового файла.  
 Листинг 52 – Программа для загрузки текстового файла

```

program TextList;
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Lists in 'Lists.pas';

//вспомогательные подпрограммы
//печать данных узла на экран
procedure NodePrint(P:Pointer);
begin
  if p<>nil then writeln(ShortString(P^));
end;

//удаление целочисленных динамических переменных
procedure DestroyData(P:Pointer);
begin
  if p<>nil then FreeMem(P);
end;

//основная программа
var F:TextFile;
    pStr:pointer;
    Buf:string[255];
    Head:PNode=nil; //список
    CurNode, Node:PNode; //текущий узел
    FileName:string;
begin
  writeln('Load TextFile');
  write('FileName: ');
  readln(FileName);

  AssignFile(F, FileName);
  {$I-}
  Reset(F);
  if IOResult<>0 then halt(1);

```



```

{$I+}
while not EOF(F) do
begin
  //считываем строку из файла во временную переменную
  readln(F, Buf);

  {выделяем память под строку
  длина строки + 1 байт}
  GetMem(pStr, Length(Buf)+1);
  {копируем строку из временной переменной
  в динамическую}
  Move(Buf, pStr^, Length(Buf)+1);

  Node:=NewNode(pStr); //создаем новый узел
  //вставляем узел в список
  if Head=nil then CurNode:=InsertBefore(Head, Node)
  else CurNode:=InsertBefore(CurNode, Node)
end;

CloseFile(F);

//вывод содержимого списка на экран
ProcessList(Head, NodePrint);

//уничтожение списка
DestroyList(Head, DestroyData);
writeln('Press Enter to exit');
readln;
end.

```

В приведенной программе для копирования строки использована процедура Move из модуля System. Эта процедура предназначена для копирования данных из одной области памяти в другую и имеет следующий синтаксис.

**procedure** Move(**const** Source; **var** Dest; Count: Integer);

где Source – источник данных (что копировать);

Dest – приемник данных (куда копировать);

Count – объем копируемых данных в байтах (сколько копировать).

## Задания к лабораторной работе

1. Изучите все подпрограммы модуля Lists и объясните алгоритм их работы. (По согласованию с преподавателем).
2. Наберите и изучите алгоритмы программ приведенных в лабораторной работе.
3. Напишите функцию Last для нахождения последнего элемента списка. Добавьте ее в модуль Lists. Напишите пример программы использующий эту функцию.
4. Перепишите программу из листинга 2 лабораторной работы №5 с использованием односвязных списков. Добавьте в программу следующие возможности:
  - удаление произвольной записи из таблицы;
  - сортировка таблицы по столбцу фамилия;
  - поиск записи по фамилии, причем если число таких записей более одной, то должны выводиться все записи;

- вывод только студентов заданной группы;
- сохранение списка в типизированный файл;
- загрузка списка из типизированного файла.

Поместите основные подпрограммы в модуль Tables. Реализуйте меню. Каждую функцию программы реализуйте в виде отдельной подпрограммы.

## Вопросы к лабораторной работе

1. Что такое односвязный список?
2. Что такое узел? Как описывается узел односвязного списка?
3. Объясните алгоритм вставки узла в односвязный список.
4. Объясните алгоритм удаления узла из односвязного списка.
5. Объясните алгоритм сортировки вставками.
6. Объясните алгоритм прохождения односвязного списка.
7. Для чего используется процедура Move?

## Справочные таблицы

### Приложение А – Модуль Lists

Листинг 53 – Модуль для работы с односвязными списками

```

unit Lists;

interface
type PNode=^TNode;

    //структура описывающая узел односвязного списка
    TNode=record
        next:PNode; //указатель на следующий узел
        data:pointer; //указатель на данные
    end;

    {подпрограмма для обработки данных узла
    ВХОДНЫЕ ПАРАМЕТРЫ
    P - указатель на данные узла}
    TProcessNode = procedure (P:Pointer);

    {функция для сравнения двух узлов
    ВХОДНЫЕ ПАРАМЕТРЫ
    P1, P2 - указатели на данные узлов

    Результат функции
    1 - если P1>P2
    -1 - если P1<P2
    0 - если P1=P2
    }
    TCompareProc = function (P1, P2:Pointer):ShortInt;

    {возвращает указатель на вновь созданный узел
    ВХОДНЫЕ ПАРАМЕТРЫ
  
```

```

P - указатель на данные узла}
function NewNode(Data:pointer):PNode;

{вставка узла InsNode после узла ANode
ВХОДНЫЕ ПАРАМЕТРЫ
  ANode   - узел, после которого необходимо вставить новый узел
  InsNode - вставляемый узел

Результат функции
  возвращается указатель на вставленный узел}
function InsertBefore(var ANode:PNode; InsNode:PNode):PNode;

{удаление узла после узла ANode,
ВХОДНЫЕ ПАРАМЕТРЫ
  ANode   - узел, после которого необходимо удалить узел
Результат функции
  возвращает указатель на удаленный узел}
function DeleteBefore(ANode:PNode):PNode;

{удаление заданного узла из списка,
ВХОДНЫЕ ПАРАМЕТРЫ
  AHead - первый узел списка
  ANode  - удаляемый узел
Результат функции
  возвращает указатель на удаленный узел}
function DeleteNode(var AHead:PNode; ANode:PNode):PNode;

{переход к следующему узлу
ВХОДНЫЕ ПАРАМЕТРЫ
  ANode - узел списка
Результат функции
  указатель на следующий узел}
function Next(ANode:PNode):PNode;

{переход к предыдущему узлу
ВХОДНЫЕ ПАРАМЕТРЫ
  ANode - узел списка
Результат функции
  указатель на предыдущий узел}
function Prev(AHead, ANode:PNode):PNode;

{удаление списка
ВХОДНЫЕ ПАРАМЕТРЫ
  AHead - первый узел списка
  ANodeProc - процедура отвечающая за удаление данных узла,
             если данные не нужно удалять, то необходимо в
             качестве параметра передать nil}
procedure DestroyList(var AHead:PNode; ANodeProc:TProcessNode);

{обработка элементов списка
ВХОДНЫЕ ПАРАМЕТРЫ
  AHead - первый узел списка
  ANodeProc - процедура отвечающая за обработку узла

Выполняется обход всех элементов списка с вызовом для
каждого элемента процедуры ANodeProc}
procedure ProcessList(AHead:PNode; ANodeProc:TProcessNode);

{поиск минимального элемента

```

**ВХОДНЫЕ ПАРАМЕТРЫ**

AHead - первый узел списка  
CompareProc - функция сравнения узлов

Результат функции  
указатель на "минимальный" узел}

**function** SearchMin(AHead:PNode; CompareProc:TCompareProc):PNode;

{поиск максимального элемента  
**ВХОДНЫЕ ПАРАМЕТРЫ**  
AHead - первый узел списка  
CompareProc - функция сравнения узлов

Результат функции  
указатель на "максимальный" узел}

**function** SearchMax(AHead:PNode; CompareProc:TCompareProc):PNode;

{сортировка списка методом вставок  
**ВХОДНЫЕ ПАРАМЕТРЫ**  
AHead - первый узел сортируемого списка  
CompareProc - функция сравнения узлов

**ВЫХОДНЫЕ ПАРАМЕТРЫ**  
AHead - первый узел отсортированного списка}

**procedure** InsertionSort(var AHead:PNode;  
CompareProc:TCompareProc);

{поиск элемента списка  
**ВХОДНЫЕ ПАРАМЕТРЫ**  
AHead - первый узел списка  
data - указатель на некоторые данные используемые  
параметром CompareProc, если данные не нужны, то nil  
CompareProc - функция сравнения узлов, для узла  
соответствующего критерию поиска должна  
возвращать 0

Результат функции  
указатель на найденный узел или nil если узел не найден}

**function** SearchNode(AHead:PNode; data:Pointer;  
CompareProc:TCompareProc):PNode;

{поиск нескольких элементов списка соответствующих критерию  
поиска  
**ВХОДНЫЕ ПАРАМЕТРЫ**  
AHead - первый узел списка  
data - указатель на некоторые данные используемые параметром  
CompareProc,  
если данные не нужны, то nil  
CompareProc - функция сравнения узлов, для узла  
соответствующего критерию поиска должна  
возвращать 0

Результат функции  
указатель на список найденных узлов или nil если  
не найден ни один узел

При формировании результата функцией SearchNodes создается  
новый список, причем указатели на данные копируются в новый  
список. При удалении списка с результатами поиска процедурой  
DestroyList целесообразно в качестве параметра ANodeProc этой

подпрограммы использовать nil. Иначе данные содержащиеся в исходном списке могут быть уничтожены!}

```
function SearchNodes(AHead:PNode; data:Pointer;  
CompareProc:TCompareProc):PNode;
```

implementation

```
function NewNode(Data:pointer):PNode;  
begin  
  Result:=new(PNode);  
  Result^.data:=Data;  
  Result^.next:=nil;  
end;{NewNode}
```

```
function InsertBefore(var ANode:PNode; InsNode:PNode):PNode;  
begin  
  //если список не содержит узлов  
  if ANode=nil then ANode:=InsNode  
  else //если вставляемый узел не пустой  
    if InsNode<>nil then  
      begin  
        InsNode^.next:=ANode^.next;  
        ANode^.next:=InsNode;  
      end;{if InsNode}  
    Result:=InsNode;  
  end;{InsertBefore}
```

```
function DeleteBefore(ANode:PNode):PNode;  
begin  
  if ANode<>nil then  
    if ANode^.next<>nil then //если удаляемый узел существует  
      begin  
        Result:=ANode^.next;  
        ANode^.next:=ANode^.next^.next;  
        Result^.next:=nil;  
      end{if ANode^.next}  
    else Result:=nil  
  else Result:=nil;  
end;{DeleteBefore}
```

```
function Next(ANode:PNode):PNode;  
begin  
  if ANode<>nil then Result:=ANode^.next  
  else Result:=nil;  
end;{Next}
```

```
function DeleteNode(var AHead:PNode; ANode:PNode):PNode;  
var CurNode:PNode;  
begin  
  Result:=nil;  
  if AHead<>ANode then  
    begin  
      CurNode:=AHead;  
      while CurNode<>nil do //поиск узла для удаления  
        begin  
          if CurNode^.next=ANode then  
            begin  
              result:=ANode;  
              //узел найден, удаляем  
              DeleteBefore(CurNode);
```

```

        break;
    end;{if}
    CurNode:=Next(CurNode);
end;{while}
end
else
begin
    AHead:=Next(AHead);
    Result:=ANode;
    ANode^.next:=nil;
end;
end;{DeleteNode}

function Prev(AHead, ANode:PNode):PNode;
begin
    Result:=nil;
    //поиск узла указатель next которого равен ANode
    while AHead<>nil do
        begin
            if AHead^.next=ANode then
                begin
                    Result:=AHead;
                    break;
                end;{if}
            AHead:=Next(AHead);
        end;{while}
    end;{Prev}

procedure DestroyList(var AHead:PNode; ANodeProc:TProcessNode);
var CurNode:PNode;
begin
    //начало списка
    CurNode:=AHead;
    //обход списка начиная с первого узла
    while CurNode<>nil do
        begin
            //следующий узел
            CurNode:=Next(CurNode);
            //если задана процедура удаления данных узла, то выполняем ее
            if Assigned(ANodeProc) then
                begin
                    ANodeProc(AHead^.data);
                    AHead^.data:=nil;
                end;{if}
            //удаляем узел
            dispose(AHead);
            //запоминаем текущий узел
            AHead:=CurNode;
        end;{while}
    end;{DestroyList}

procedure ProcessList(AHead:PNode; ANodeProc:TProcessNode);
begin
    //обход списка начиная с первого узла
    while AHead<>nil do
        begin
            ANodeProc(AHead^.data); //процедура обработки данных узла
            AHead:=Next(AHead);
        end;{while}
    end;{ProcessList}

```

```

function SearchMin(AHead:PNode; CompareProc:TCompareProc):PNode;
var minNode:PNode;
begin
    minNode:=AHead;
    AHead:=Next(AHead);
    //обход списка начиная со второго узла
    while AHead<>nil do
        begin
            if CompareProc(minNode^.data, AHead^.data)>=0 then
minNode:=AHead;
            AHead:=Next(AHead);
        end;{while}
    Result:=minNode;
end;{SearchMin}

function SearchMax(AHead:PNode; CompareProc:TCompareProc):PNode;
var minNode:PNode;
begin
    minNode:=AHead;
    AHead:=Next(AHead);
    //обход списка начиная со второго узла
    while AHead<>nil do
        begin
            if CompareProc(minNode^.data, AHead^.data)<=0 then
minNode:=AHead;
            AHead:=Next(AHead);
        end;{while}
    Result:=minNode;
end;{SearchMax}

function SearchNode(AHead:PNode; data:Pointer;
CompareProc:TCompareProc):PNode;
begin
    Result:=nil;
    //обход списка начиная с первого узла
    while AHead<>nil do
        //проверка узла на соответствие критерию поиска
        if CompareProc(AHead^.data, data)=0 then
            begin
                Result:=AHead;
                break;
            end
        else AHead:=Next(AHead);
    end;{SearchNode}

function SearchNodes(AHead:PNode; data:Pointer;
CompareProc:TCompareProc):PNode;
var TempList, Node:PNode;
begin
    TempList:=nil; //список найденных узлов

    //обход списка, начиная с первого узла
    while AHead<>nil do
        begin
            //проверка узла на соответствие критерию поиска
            if CompareProc(AHead^.data, data)=0 then
                begin
                    //создаем новый узел, копируем указатель на данные узла
                    Node:=NewNode(AHead^.data);

```

```

    //вставка узла в список найденных узлов
    InsertBefore(TempList, Node);
end;{if}
AHead:=Next(AHead);
end;{while AHead}

Result:=TempList;
end;{SearchNodes}

procedure InsertionSort(var AHead:PNode;
CompareProc:TCompareProc);
var TempList:PNode;
    CurNode, Cur, PrevNode:PNode;
begin
    {создаем список, содержащий минимальный элемент исходного
списка}
    TempList:=SearchMin(AHead, CompareProc);
    //удаляем найденный минимальный элемент из исходного списка
    TempList:=DeleteNode(AHead, TempList);
    //обход исходного списка начиная с первого элемента
    while AHead<>nil do
        begin
            {удаляем текущий элемент из исходного списка и
переходим к следующему элементу}
            Cur:=AHead;
            AHead:=AHead^.next;
            Cur^.next:=nil;
            CurNode:=TempList; //установка на начало списка
            PrevNode:=CurNode;

            while CurNode<>nil do //поиск места вставки
                begin
                    if CompareProc(CurNode^.data, Cur^.data)>=0 then
                        begin
                            {если вставляемый элемент "меньше" текущего, то
вставляем его}
                            InsertBefore(PrevNode, Cur);
                            break;
                        end{if}
                    else
                        begin
                            PrevNode:=CurNode; //запоминаем текущий узел
                            CurNode:=Next(CurNode); //переходим к следующему узлу
                        end{else}
                    end;{ while CurNode}

                    {вставка элемента в конец списка}
                    if CurNode=nil then InsertBefore(PrevNode, Cur);
                end;{while AHead}

                //указатель на отсортированный список
                AHead:=TempList;
            end;{InsertionSort}
        end;{Lists}

```



# Лабораторная работа №10

## Основы объектно-ориентированного программирования

### Введение

В данной лабораторной работе рассмотрены основные принципы объектно-ориентированного программирования. Рассмотрена объектная модель языка Object Pascal, некоторые стандартные классы Object Pascal. Приведены примеры программ.

### Основные понятия ООП

Первые программы для цифровых вычислительных машин редко превышали объем 1 Кбайт. С той поры существенно изменилась архитектура и технические характеристики программируемых вычислительных средств. Объемы используемых программ и программных систем измеряются не только десятками килобайтов, но и сотнями мегабайтов. Было обнаружено, что время создания сложных программ пропорционально квадрату или даже кубу объема программ. Поэтому одним из основных факторов, определяющих развитие технологии программирования, является снижение стоимости проектирования и создания программных продуктов, или борьба со сложностью программирования.

В программировании широко используется фундаментальный принцип управления сложными системами, согласно которому при проектировании сложной технической системы проводится декомпозиция решаемой задачи. Целью декомпозиции является представление разрабатываемой системы в виде взаимодействующих небольших подсистем (модулей или блоков), каждую из которых можно отладить (испытать) независимо от других.

Идеи разделения программ на относительно самостоятельные крупные части, реализующие определенные процедуры и функции и образующее определенную иерархию взаимосвязей, нашли отражение в **структурном подходе** к разработке и созданию программных средств. В программировании структурный подход появился с возникновением первых подпрограмм, процедур и функций, написанных в так называемом **процедурно ориентированном стиле**. Данный стиль опирается на простое правило: определить переменные и константы, которые понадобится хранить в памяти компьютера, и описать или использовать алгоритм их обработки.

Дальнейшее развитие структурного подхода привело к **модульному программированию**, оно предусматривает декомпозицию прикладной задачи в виде иерархии взаимодействующих модулей или программ. Модуль, содержащий данные и процедуры их обработки удобен для автономной разработки и создания.

Введение типов данных обозначило еще одно направление развития технологии программирования. Типизация данных предназначена для облегчения составления программ, так и для автоматизации выявления ошибок использования данных в виде операндов и фактических параметров при вызове функций.

Результатом обобщения понятия «тип данных» являются классы объектов, которые могут содержать в качестве элементов не только данные определенного типа, но и методы их обработки (функции и процедуры). Таким образом, развитие идей абстрагирования и модульности привело к появлению в программировании объектного подхода.

Человек мыслит образами или объектами, он знает их свойства и манипулирует ими, сообразуясь с определенными событиями. Так, подумав о телефонном аппарате, человек может представить не только его форму и цвет, но и возможность позвонить, характер звучания звонка и ряд других свойств.

Развитием идеи модульного программирования является сопоставление объектам предметной области (моделируемым объектам) программных конструкций, называемых объектами, объектными типами или классами (моделирующими объектами).

Моделирующие объекты содержат данные и подпрограммы, которые описывают свойства моделируемых объектов. Так, данные могут отражать признаковые или количественные свойства (масса, длина, мощность, цена, наличие, видимость и т. п.), а подпрограммы отражают поведенческие или операционные свойства (изменить массу, вычислить мощность, установить цену, проверить наличие, сделать видимым и т. п.). Таким образом, при объектном подходе интеграция данных и процедур их обработки определяется структурой предметной области, т.е. набором моделируемых объектов, их взаимосвязью или взаимодействием в рамках решаемой задачи.

Моделируемый объект всегда представляется человеку чем-то единым, целостным, хотя может состоять из частей или других объектов. Например, телефонный аппарат состоит из трубки, провода, корпуса с номеронабирателем, а трубка содержит микрофон, динамик и провода и т. д. Целостное представление объекта в виде взаимосвязанной совокупности его свойств или компонентов является базовым принципом объектного подхода.

В настоящее время **объектно-ориентированное программирование** (ООП) является доминирующим стилем при создании больших программ.

Основным понятием ООП является **класс**, который можно рассматривать с двух позиций. Во-первых, с позиции предметной области: класс соответствует определенному характерному (типичному) объекту этой области. Во-вторых, с позиции технологии программирования, реализующей это соответствие: «класс» в ООП – это определенная программная структура, которая обладает тремя важнейшими свойствами:

- инкапсуляции;
- наследования;
- полиморфизма.

Эти свойства используются программистом, а обеспечиваются объектно-ориентированным языком программирования (транслятором, реализующим этот язык). Они позволяют адекватно отражать структуру предметной области.

## **Объекты и классы**

Концепция классов предназначена для моделирования (отображения) понятий предметной области в виде программных единиц, объединяющих в себе атрибуты и поведение (состояние и функционирование) соответствующих объектов (сущностей) предметной области.

Класс объектов представляет собой программную структуру, в которой данные и функции образуют единое целое и отражают свойства и поведение этого целого в рамках моделируемой предметной области. В отличие от модуля, в котором на состав данных и функций накладывается меньше смысловых ограничений, в объекте присутствуют только те данные и функции, которые необходимы для описания свойств и поведения объекта определенного типа.

Классы выделяются в процессе анализа предметной области с использованием идей абстрагирования от несущественного и классификации родственных объектов. Результатом объектно-ориентированного анализа являются классы объектов, которые присутствуют или в перспективе могут присутствовать в пространстве решаемой задачи и образуют иерархии классов, представляемые в виде деревьев наследования свойств.

Класс объектов характеризуется уникальным набором свойств и ему присваивается уникальное имя, как и любому типу данных. В качестве переменных программы используются объекты определенного класса. Создаваемые объекты, даже одного класса, могут отличаться значениями (степенью проявления) свойств и, конечно, отличаются именами.

Здесь следует уточнить различие понятий объект и класс. Под объектом понимается какой-либо конкретный экземпляр класса. Например, если рассматривать класс дерева, то под объектом будем понимать какое-то конкретное дерево, входящее в этот класс.

## **Инкапсуляция**

**Инкапсуляция** (дословно – «содержание в оболочке») представляет собой объединение и локализацию в рамках объекта, как единого целого, данных и функций, обрабатывающих эти данные. В совокупности они отражают свойства объекта.

В **Object Pascal** данные класса и объекта называются полями, а функции и процедуры – методами. Доступ к полям и методам объекта осуществляется через имя объекта и соответствующие имена полей и методов при помощи операций выбора «.». Хорошим стилем в ООП считается организация доступа к полям только с помощью методов. Это позволяет в максимальной степени изолировать содержание объекта от внешнего окружения, т.е. ограничить и наглядно контролировать доступ к элементам объекта. В результате замена или модификация полей и методов, инкапсулированных в объект, как правило, не влечет за собой плохо контролируемых последствий для программы в целом.

Для обеспечения лучшего контроля доступа к полям, а также методам, в каждом поле или методу можно сопоставить следующие атрибуты доступа: **private**, **protected** и **public**. Соответственно, доступен только внутри класса, только потомкам класса, общедоступен. Гибкое разграничение доступа позволяет уменьшить нежелательные (бесконтрольные) искажения свойств объекта или несанкционированное использование свойств классов.

## **Наследование**

**Наследование** – это способность классов к порождению потомков и передачи им своих свойств. Класс-потомок автоматически наследует от родителя все поля и методы, а также может содержать новые поля и методы и даже заменять методы родителя или модифицировать их. Наследование в ООП позволяет адекватно отражать родственные отношения объектов предметной области.

Если класс В обладает всеми свойствами класса А и еще имеет дополнительные свойства, то класс А называется базовым, а класс В называется наследником класса А.

Свойство наследования упрощает модификацию свойств классов, обеспечивает ООП исключительную гибкость и сокращает затраты на написание новых классов на основе уже созданных. Обычно определяется некоторый базовый класс, обладающий наиболее общими свойствами, а затем создается последовательность потомков, которые обладают своими специфическими свойствами. В результате получается иерархия наследования свойств классов.

Базовые классы, обычно, обладают такими абстрактными свойствами, что непосредственно в программах не используются, а необходимы для порождения требуемых классов. Правильный выбор базового класса обеспечивает удобство создания библиотеки классов.

В качестве примера рассмотрим иерархию классов геометрических примитивов приведенную на рисунке 1.

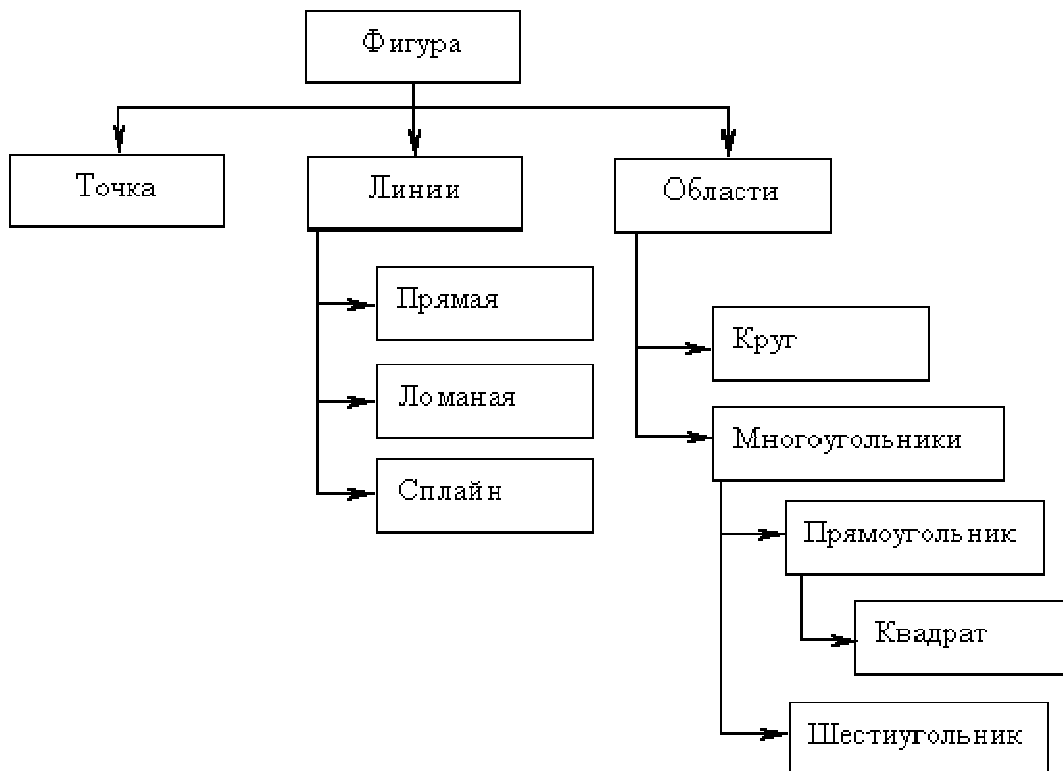


Рисунок 21 – Иерархия классов геометрических примитивов

Как видно из рисунка, базовым для всех классов является класс «Фигура». Очевидно, что этот класс будет содержать мало кода связанного с отображением или расчетом характеристик геометрических примитивов, но он будет содержать описание основных методов используемых всеми его потомками, которые будут перекрываться в них.

## Полиморфизм

**Полиморфизм** – это свойство объектов-родственников по-разному осуществлять однотипные (и даже одинаково поименованные) действия, т.е. однотипные действия у множества родственных классов имеют множество различных форм. Например, метод «нарисовать на экране» должен по-разному реализовываться для родственных классов «точка», «прямая», «ломаная», «прямоугольник». В ООП действия или поведение объекта определяется набором методов. Изменяя алгоритм метода в потомках класса, программист придает наследникам специфические поведенческие свойства. Для изменения метода необходимо перекрыть его в потомке.

Свойство полиморфизма может быть реализовано не только в механизме замещения одноименных методов при наследовании свойств, но и с помощью механизма позднего связывания методов.

Если используется **механизм раннего связывания**, то замещение метода реализуется на этапе компиляции. Так как вся работа по замещению одного метода другим происходит на этапе компиляции, то вызов метода происходит также как вызов обычной подпрограммы, т.е. для вызова не требуется никаких дополнительных ресурсов.

Если используется **механизм позднего связывания**, то замещение метода реализуется во время работы программы. Механизм позднего связывания используется тогда, когда невозможно осуществить ранее связывание. Эта проблема возникает, если на этапе компиляции нельзя определить тип объекта (класс к которому принадлежит объект). Методы, использующие позднее связывание называют виртуальными. Так как работа по замещению методов при позднем связывании происходит при каждом вызове метода, то

виртуальные методы будут выполняться медленнее, из-за необходимости дополнительных действий по связыванию.

Кроме механизмов раннего и позднего связывания полиморфизм может быть реализован с помощью **механизма делегирования метода**. Механизм делегирования метода состоит в передаче какого-либо метода объекта в качестве параметра, подобно передаче в качестве параметра подпрограммы. В отличие от механизмов раннего и позднего связывания, которые изменяют поведение всего класса, делегирование позволяет изменить поведение отдельного экземпляра класса (объекта).

## Объектная модель Object Pascal

Классами в **Object Pascal** называются специальные типы, которые содержат элементы трех видов, а именно: поля, методы и свойства. Как и любой тип, класс служит лишь образцом для создания конкретных экземпляров, которые называются объектами.

Следует отметить, что переменные объектного типа всегда располагаются в динамической памяти и фактически представляют собой указатели. Как и при использовании динамических массивов и длинных строк оператор разадресации указателя при работе с классами не применяется<sup>7</sup>, т.к. компилятор автоматически производит все необходимые действия.

Обратите внимание, что при присвоении переменных объектного типа друг другу возникает та же проблема, что и при использовании динамических массивов, т.е. происходит лишь копирование указателей на объекты, а не данных содержащихся в объектах.

### Объявление класса

Классы описываются в **Object Pascal** в секции **type**. Синтаксис описания класса имеет следующий вид

```
<имя класса>=class([имя предка])
```

```
private
```

```
<элементы класса>
```

```
protected
```

```
<элементы класса>
```

```
public
```

```
<элементы класса>
```

```
end;
```

```
здесь
```

```
<имя класса> – правильный идентификатор;
```

```
class – зарезервированное слово, открывает описание класса;
```

```
<имя предка> – имя родительского класса, если не указано, то родительским считается класс TObject;
```

**private**, **protected**, **public** – секции определяющие соответствующие уровни доступа к элементам класса: собственный, защищенный и общедоступный;

---

<sup>7</sup> Более того, использование оператора разадресации при работе с объектами запрещено.

<элементы класса> – список состоящий из трех видов элементов: полей, методов, свойств.

Именованние классов, полей, методов и свойств имеет большое значение в **Object Pascal**. Названия должны либо совпадать с названиями, использующимися в предметной области, либо ясно отражать смысл или назначение именуемого класса, поля или метода. При этом не следует бояться длинных имен – затраты на написание окупятся при отладке и сопровождении программы. Кроме того, в **Object Pascal** принято применять префиксы для идентификаторов различного назначения, например, идентификаторы почти всех типов, в том числе и классов начинаются с большой буквы «Т», а наименование полей классов с «F».


Любой класс **Object Pascal** может содержать три секции, а именно: **private** (личные), **protected** (защищенные), **public** (доступные). Внутри каждой секции определяются поля, а затем методы и свойства. Секции определяют области видимости элементов описания класса.

**Секция public** не накладывает ограничений на область видимости перечисляемых в ней полей, методов, свойств – их можно вызывать в любом модуле программы. По умолчанию если секция не указана, то считается, что объявлена секция **public**.

**Секция private** максимально сужает область видимости. Элементы класса, описанные в этой секции, доступны только внутри методов данного класса и подпрограммах объявленных в этом же модуле. Элемент класса, описанный в секции **private**, не доступен даже ближайшим потомкам класса, если они описаны в других модулях.

**Секция protected.** Элемент класса, описанный в этой секции, доступен только методам самого класса, а также внутри методов его потомков, вне зависимости от того в каком модуле они описаны.

В **Object Pascal** разрешено сколько угодно раз объявлять любую секцию, причем порядок следования секций не имеет значения. Любая секция может быть пустой.

 **Совет:** Для ускорения набора текста программы можно воспользоваться возможностью вставки шаблонов кода в среде **Delphi**. Для этого необходимо нажать следующее сочетание клавиш **Ctrl+J**. Появится окно выбора шаблона, после чего необходимо выбрать нужный шаблон и нажать **Enter**. Ниже приведен пример вставки заготовки класса из шаблона.

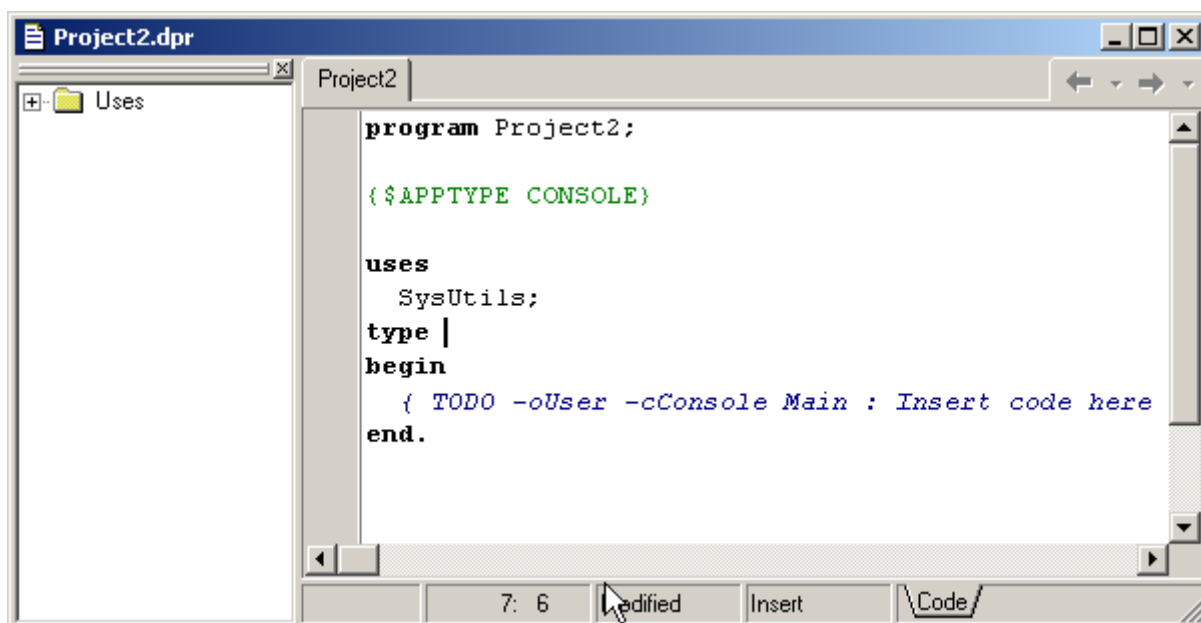


Рисунок 22 – На место где установлен курсор будет добавлен шаблон

Теперь необходимо нажать Ctrl+J. Появится окно выбора шаблона.

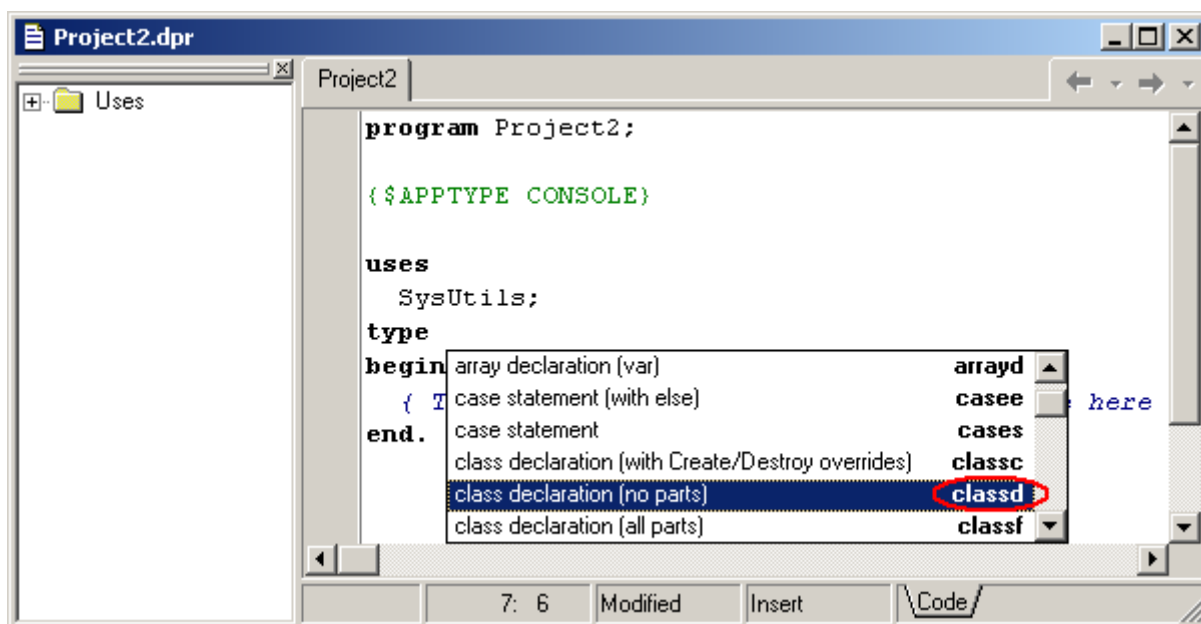


Рисунок 23 – Выбор шаблона для вставки

Выбираем шаблон для вставки и нажимаем Enter. Для более быстрой работы можно применить другой метод вставки шаблона кода а именно напечатать имя шаблона (обведено красным на рисунке 3), установить на него курсор и нажать Ctrl+J, тогда соответствующий шаблон будет вставлен автоматически без вывода окна выбора шаблона.

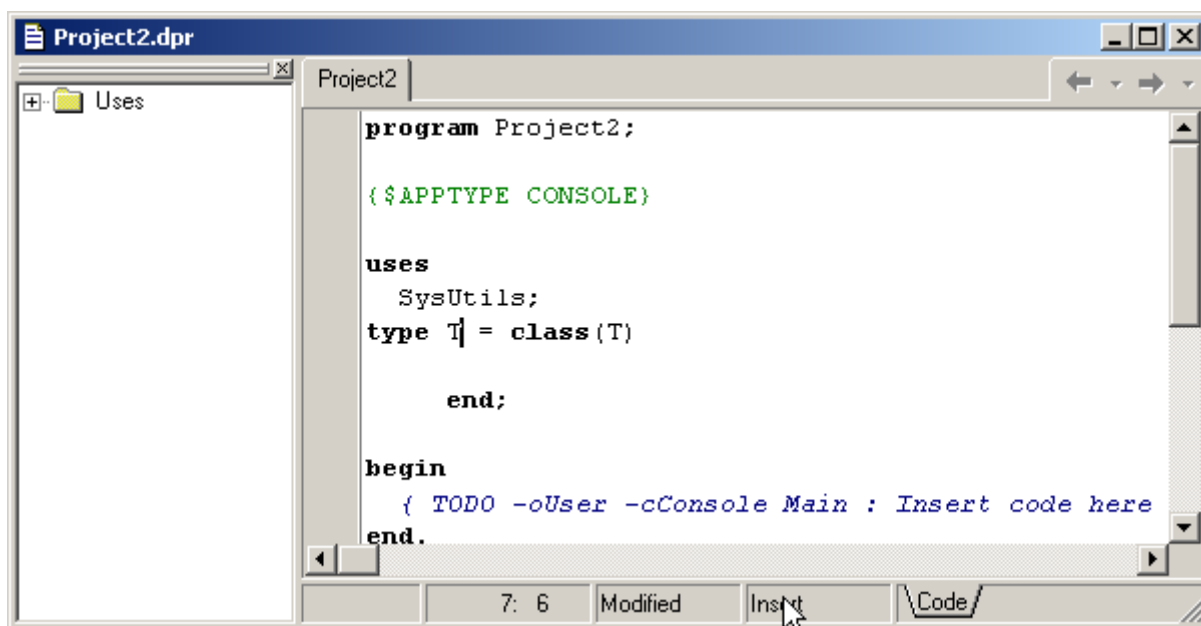


Рисунок 24 – Вставлен шаблон для описания класса

После того как шаблон вставлен его, необходимо заполнить, напечатав имя класса, поля, ...

## Элементы класса

К элементам класса относятся поля, методы и свойства. В полях хранятся данные. Методы реализуют алгоритмы обработки данных класса. Свойства предназначены для организации более гибкого механизма доступа к полям и методам класса. Механизм свойств работает только на этапе компиляции программы и (обращение к свойствам на этапе компиляции программы заменяется вызовами соответствующих методов или

обращением к полям), следовательно, не влияет на эффективность работы программы. Программы при применении свойств становятся более удобочитаемыми и менее подверженными ошибкам.

## Поля

Поля классов служат для хранения данных. Поля могут содержать данные любых типов, включая классы. Каждый объект получает уникальный набор полей, но общий для всех объектов класса набор методов и свойств. Синтаксис определения полей совпадает с синтаксисом определения переменных и имеет следующий вид

```
<id>, <id>, ...<id>:<имя типа>;
```

В **Object Pascal** имена полей принято начинать с «F» (Field). Для обращения к полю необходимо применять оператор «.», например

Листинг 54

```
type TFigure=class
    Fname:string[23];
    Fid:Integer;
    ..
end;
var Fig:TFigure;
begin
    Fig.Fname:='line';
end.
```

В соответствии с принципами ООП доступ к полям должен осуществляться только с помощью методов, при таком доступе к полям наиболее полно реализуется концепция инкапсуляции данных и внутренних структур объекта. **Object Pascal** может использовать более гибкий механизм свойств для реализации инкапсуляции. Более подробно механизм свойств будет описан ниже. Пока можно лишь отметить, что поля, как правило, размещают в секции **private** и никогда не изменяют напрямую.

## Методы

Инкапсулированные в классе подпрограммы называются методами класса. Синтаксис объявления методов совпадает с синтаксисом объявления процедур и функций, например

Листинг 55

```
type TFigure=class
    private
        Fname:string[23]; //поля
        Fid:Integer;
    public
        procedure SetName(Value:string); //методы
        function GetName:string;
    end;
procedure TFigure.SetName(Value:string);
begin
    Fname:=value;
end;
function TFigure.GetName(Value:string);
begin
    Result:=Fname;
```



```

end;
var Fig:TFigure;
begin
  Fig.SetName('line');
  WriteLn(Fig.GetName);
end.

```

Доступ к методам осуществляется также как и к полям с использованием оператора «.»». Обратите внимание на то, что в описании класса находятся только заголовки методов (см. листинг 2). Тела методов находятся после описания класса там, где обычно описываются подпрограммы. Если класс размещается в модуле, то тела его методов, вне зависимости от того, где находится описание класса, всегда находятся в секции реализации (**implementation**). Идентификаторы методов класса имеют следующий синтаксис

<имя класса>.<заголовков метода>



**Совет:** Для ускорения набора текста программы можно воспользоваться технологией **Code Completion** среды **Delphi**. Для этого необходимо установить курсор в строку с описанием класса и нажать **Ctrl+Shift+C**. После этого система автоматически добавит заготовки для тел методов (в примере это методы *SetName* и *GetName*). Для быстрого перемещения между телом метода и его описанием можно использовать следующее сочетание клавиш **Ctrl+Shift+↑**.

## Перекрытие методов

Перекрытие метода это замена некоторого метода родительского класса другим методом в дочернем классе. Замена метода может происходить как на этапе компиляции (раннее связывание), так и во время выполнения программы (позднее связывание).

### Раннее связывание

Если замена одного метода другим происходит на этапе компиляции, то используется механизм раннего связывания. В этом случае вызов метода происходит также как и обычной подпрограммы. Условием использования механизма **раннего связывания** является, то, что **при компиляции должен быть известен тип объекта** (класс к которому он принадлежит). Для реализации раннего связывания в Object Pascal необходимо просто заново объявить перекрываемый метод в классе потомке, например

Листинг 56

```

type TFigure=class
  private
    FName:string[23]; //поля
    Fid:Integer;

  public
    procedure SetName(Value:string); //методы
    function GetName:string;
    //процедура выводящая информацию об объекте
    procedure Draw;
  end;

  //дочерний класс
  TDot=class(TFigure)
  private
    Fx, Fy:Integer; //поля координаты точки

```

```

    public
    //методы для доступа к полям Fx, Fy
    procedure SetX(Value:Integer);
    procedure SetY(Value:Integer);
    function GetX(Value:Integer):Integer;
    function GetY(Value:Integer):Integer;

    procedure Draw; //перекрываем метод
end;

...
//метод Draw класса TFigure
procedure TFigure.Draw;
begin
    writeln('class: TFigure',
           'name :', GetName);
end;

//метод Draw класса TDot
procedure TDot.Draw;
begin
    writeln('class: TDot',
           'name :', GetName,
           'x: ', GetX,
           'y: ', GetY);
end;

//основная программа
var Fig:TFigure;
    Dot:TDot;
begin
    ...
    Fig.Draw; //вызов метода TFigure.Draw
    Dot.Draw; //вызов метода TDot.Draw
    ...
end.

```

В примере приведенном в листинге 3 введен новый дочерний класс от TFigure класс TDot (точка). Класс TDot добавляет к элементам родительского класса два новых поля и четыре новых метода для доступа к полям. Также в TDot перекрывается метод Draw родительского класса. Из листинга 3 хорошо видно, что компилятор легко определит тип объекта (см. основную программу) при вызове метода Draw и вставит вызов соответствующего экземпляра метода TFigure.Draw или TDot.Draw.

### *Позднее связывание*

Введем в класс TFigure метод Show, листинг 4

Листинг 57

```

type TFigure=class
    private
    public
    ...
    procedure Draw;
    procedure Show;
end;

//дочерний класс
TDot=class(TFigure)

```

```

    ...
    procedure Draw;
end;

...
procedure TFigure.Show;
begin
    writeln('class Info');
    Draw;
end;

//основная программа
var Fig:TFigure;
    Dot:TDot;
begin
    ...
    Fig.Show; //вызов метода TFigure.Draw
    Dot.Show; //вызов метода TDot.Draw
end.

```

В листинге 4 приведен типичный пример кода, когда использование механизма раннего связывания не приводит к требуемому результату. В результате выполнения программы на экран будет выведено две одинаковые строки. В обоих случаях будет выведена информация о классе TFigure. Причиной такого поведения программы является то, что компилятор не может правильно определить тип объекта вызывающего метод Draw в теле метода Show.

Рассмотрим подробнее тело метода Show. Метод Show описывается в классе TFigure, при компиляции тела метода Show компилятору ничего не известно о том какой родительский или дочерний класс его будет вызывать. Встречая метод Draw в теле метода Show, компилятор просто вставляет вызов подпрограммы TFigure.Draw. Класс TDot наследует метод Show, поэтому при выполнении метода TDot.Show в теле метода вызывается не TDot.Draw, а TFigure.Draw.

Для решения указанной проблемы используется механизм позднего связывания методов. Суть метода состоит в том, что перед вызовом метода компилятором вставляется специальный код, который осуществляет выбор экземпляра метода для вызова. В приведенном примере будет определяться, в каком случае необходимо вызвать TFigure.Draw, а в каком TDot.Draw.

Для указания компилятору, что для метода необходимо использовать механизм позднего связывания необходимо после заголовка метода указать директиву слово **virtual**. При перекрытии метода в дочернем классе необходимо после заголовка метода указать директиву **override**. Методы, использующие механизм позднего связывания, называют виртуальными.

Листинг 58

```

type TFigure=class
    private
    public
        ...
        procedure Draw; virtual;
        procedure Show;
    end;

//дочерний класс
TDot=class(TFigure)
    ...

```

```

    procedure Draw; override;
end;

...
procedure TFigure.Show;
begin
    writeln('class Info');
    Draw;
end;

//основная программа
var Fig:TFigure;
    Dot:TDot;
begin
    ...
    Fig.Show; //вызов метода TFigure.Draw
    Dot.Show; //вызов метода TDot.Draw
...
end.

```

Программа, приведенная в листинге 5, выдаст ожидаемый результат, а именно выведет на экран информацию об объекте Fig и Dot.

Следует помнить, что при вызове виртуальных методов имеются накладные расходы, связанные с определением экземпляра вызываемого метода, поэтому виртуальные методы всегда выполняются медленнее. Также следует помнить следующий принцип, метод объявленный виртуальным в родительском классе будет виртуальным во всех дочерних классах. Иначе говоря, **однажды виртуальный – всегда виртуальный**.

## Конструкторы и деструкторы

Каждый класс кроме обычных методов содержит два метода специального назначения. Эти методы имеют специальное наименование конструктор и деструктор.

### *Конструкторы*

Конструктор выполняет инициализацию объекта при его создании. При вызове конструктора выделяется память под поля объекта, инициализируется механизм позднего связывания, также конструктор может выполнять некоторый пользовательский код. В теле конструктора обычно выполняются действия по начальной настройке объекта, например, заполняются поля объекта некоторыми данными. В общем, конструктор мало, чем отличается от процедуры. Синтаксис описания конструктора приведен ниже

**constructor** <имя конструктора>([<список формальных параметров>]);

Как уже говорилось выше, любой класс **Object Pascal** является потомком класса TObject. Класс TObject содержит виртуальные методы и, поэтому, для создания объекта обязательно применение конструктора (для инициализации механизма позднего связывания)<sup>8</sup>. В классе TObject определен конструктор Create. Этот конструктор наследуют все порожденные классы. При необходимости можно перекрыть конструктор родительского класса. Обычно при перекрытии конструкторов используется ранее связывание. Пример конструктора для класса TFigure приведен ниже.

```

type TFigure=class
    private
    ...
    public

```

<sup>8</sup> Кроме того, в конструкторе выделяется память под объект, т.к. все объекты Object Pascal находятся в динамической памяти.

```

    ...
    procedure SetName(Value:string);
    //перекрываем конструктор родительского класса TObject
    constructor Create;
    ...
end;

...

constructor TFigure.Create;
begin
    inherited Create; //вызов унаследованного конструктора
    SetName('TFigure');
end;

```

Обратите внимание на тело конструктора, первая строка конструктора, обычно, содержит вызов унаследованного конструктора (для этого применяется зарезервированное слово **inherited**), т.к. он может содержать, например, инициализацию полей родительского объекта либо еще какой-либо код. В приведенном примере эту строку можно пропустить т.к. тело конструктора класса TObject не содержит никакого кода, но в общем случае любой конструктор следует начинать именно с вызова унаследованного конструктора, иначе объект может работать не корректно.

При вызове конструктора существует особенность, для создания экземпляра класса (объекта) вызывается конструктор класса, а не конструктор объекта. Например, следующий код иллюстрирует правильное применение конструктора.

```

...
//основная программа
var Fig:TFigure;
begin
    ...
    Fig:=TFigure.Create; //создаем объект
    Fig.Show; //вызов метода
    ...
end.

```

После создания объекта можно вызывать его методы. Обратите внимание, что **переменная объектного типа является указателем** и, перед ее использованием необходимо создать объект и поместить указатель на него в переменную. Так как конструктор тоже является методом объекта, то вызывать его для объекта нельзя т.к. он еще не создан, поэтому строка

```
Fig.Create;
```

вызовет ошибку.

Если вызвать конструктор для уже созданного объекта, то это может привести к сбросу полей объекта в начальное состояние, но в общем случае такое поведение не гарантируется, поэтому не рекомендуется вызывать конструктор для уже созданного объекта.

### *Деструкторы*

Деструктор выполняет действие обратное действию конструктора а именно освобождает память занятую объектом. Кроме того, деструктор может содержать пользовательский код. Обычно в деструктор помещают код связанный с деинициализацией объекта, а именно: закрытие открытых файлов, освобождение динамической памяти (если объект ее использовал в своей работе) и т.п. Синтаксис объявления деструктора следующий

**destructor** <имя деструктора>([<список формальных параметров>]);

В классе TObject объявлен деструктор Destroy. Следовательно, все классы наследуют этот деструктор и могут его использовать. При перекрытии родительского деструктора следует иметь в виду, что метод (деструктор) **Destroy виртуальный** и, при его перекрытии следует использовать директиву **override**. Иначе возможно некорректное уничтожение объекта. Приведем пример перекрытия деструктора

```
type TFigure=class
  private
  public
    ...
    //перекрываем деструктор родительского класса TObject
    destructor Destroy; override;
    ...
end;
...
destructor TFigure.Destroy;
begin
  //код для освобождения ресурсов занятых объектом
  ...
  //вызов унаследованного деструктора
  inherited Destroy;
end;
```

Обратите внимание на то, что вызов унаследованного деструктора происходит в последнюю очередь. Иначе возможна ситуация когда ресурсы будут высвобождены до того как объект перестанет их использовать, что приведет к ошибке и краху программы. **Вызов унаследованного деструктора обязателен!** В противном случае возможна утечка<sup>9</sup> памяти или других ресурсов программы.

Обычно напрямую деструктор не вызывают. Для уничтожения объекта используется метод Free класса TObject, например

```
...
//основная программа
var Fig:TFigure;
begin
  ...
  Fig:=TFigure.Create; //создаем объект
  Fig.Show; //работа с объектом
  Fig.Free; //уничтожение объекта
  ...
end.
```

Деструктор вызывается в теле метода Free, причем перед вызовом деструктора метод Free проверяет, не разрушен ли объект ранее (т.е. не равен ли указатель на объект **nil**), а затем если объект не разрушен, разрушает его с помощью деструктора.

## Методы класса

Как говорилось выше до создания объекта с помощью конструктора нельзя вызывать его методы, обращаться к полям и свойствам, но существует исключение из этого

<sup>9</sup> Утечка памяти – ошибка в программе, связанная с тем, что программа выделяет динамическую память под свои нужды, а затем из-за ошибок в логике программы, не возвращает ее системе. В конце концов, это приводит к постепенному замедлению работы компьютера, а при исчерпании доступного объема памяти – краху приложения.

правила. Можно создавать **методы** к которым можно обращаться **до создания объекта** – это методы класса. Для объявления метода класса нужно использовать следующий синтаксис.

```
class procedure <имя метода>([<список форм. параметров>]);
```

или

```
class function <имя метода>([<список форм. параметров>]):<тип результата>;
```

Как видно описание методов класса отличается от описания обычных методов только наличием зарезервированного слова **class** перед заголовком метода. На методы класса наложено следующее ограничение: тело метода класса не должно содержать обращений к полям, свойствам и обычным методам класса. Это ограничение естественно т.к. до создания объекта ни полей, ни обычных методов еще не существует, и обращаться просто не к чему.

Методы класса используются для работы с информацией общей для всех объектов класса, например такой метод может возвращать текстовую строку с описанием назначения класса или счетчик созданных экземпляров класса. Например

```
type TFigure=class
  private
  public
    class function GetClassInfo:string;
  end;
...
function TFigure.GetClassInfo:string;
begin
  Result:='Abstract class TFigure';
end;

//основная программа
var Fig:TFigure;
begin
  ...
  //вызываем метод класса (до создания объекта)
  writeln(Fig.GetClassInfo);
  Fig:=TFigure.Create; //создаем объект
  Fig.Show; //работа с объектом
  Fig.Free; //уничтожение объекта
  ...
end.
```

## Абстрактные методы

Часто базовые классы содержат методы тела, которых не содержат никакого кода. Например, при реализации класса TFigure в метод Draw может не содержать никакого кода, т.к. нельзя начертить фигуру незаданной формы. Для описания таких методов необходимо применять директиву **abstract**. Методы с директивой **abstract** должны быть виртуальными и обязательно должны быть перекрыты в дочерних классах. Вызов метода объявленного с директивой **abstract** приведет к ошибке. Например

```
type TFigure=class
  private
  public
```

```

    ...
    procedure Draw; virtual; abstract;
    procedure Show;
end;

//дочерний класс
TDot=class(TFigure)
    ...
    procedure Draw; override;
end;

...

procedure TDot.Draw;
begin
    writeln('name :', GetName,
           'x: ', GetX,
           'y: ', GetY);
end;

procedure TFigure.Show;
begin
    writeln('class Info');
    Draw;
end;

//основная программа
var Fig:TFigure;
    Dot:TDot;
begin
    ...
    Fig.Show; //вызов этого метода приведет к ошибке
    Dot.Show; //вызов этого метода пройдет без ошибок
    ...
end.

```

Обратите внимание на то, что хотя метод TFigure.Draw объявлен, но его тело отсутствует. При вызове метода Show класса TFigure произойдет ошибка т.к. метод Draw класса TFigure, который вызывается в теле метода Show, объявлен абстрактным.

Классы, которые содержат абстрактные методы, называются абстрактными.

## Перегрузка методов

Начиная с Delphi 4 в **Object Pascal** возможна перегрузка процедур и функций, а также методов класса. Перегрузка методов заключается в том, что можно создавать несколько одноименных методов, но с разными списками формальных параметров. Для включения механизма перегрузки методов необходимо указать директиву **overload** после его заголовка. Например:

```

type TFigure=class
    private
    public
        ...
        procedure Metod; overload;
        procedure Metod(a:integer); overload;
        procedure Metod(a:integer; r:real); overload;
    end;
    ...
procedure TFigure.Metod;

```



```
begin
  ...
end;

procedure TFigure.Metod(a:integer);
begin
  ...
end;

procedure TFigure.Metod(a:integer; r:real);
begin
  ...
end;
```

Директива **overload** включает режим сверки не только имен подпрограмм, но и их списков формальных параметров. Вызывается та подпрограмма, для которой совпало не только имя, но и список формальных и фактических параметров (если списки формальных и фактических параметров не совпали ни для одного из вариантов, то компилятор выдаст ошибку).

### Свойства

Свойства – это специальный механизм класса регулирующий доступ к полям. Синтаксис определения свойства следующий

```
property <имя свойства>:<тип свойства>   read <имя поля|имя метода>
                                           write <имя поля|имя метода>;
```

здесь **property**, **read**, **write** – зарезервированные слова (**read** и **write** зарезервированы только в контексте объявления свойства).

После **read** указывается имя поля или имя метода из которого осуществляется чтение значения свойства. Тип свойства и тип поля, из которого осуществляется чтение, должны совпадать. Если свойство считывает значение с помощью метода, то метод должен иметь следующий заголовок

```
function <имя метода>:<тип свойства>;
```


Обратите внимание на то, что тип возвращаемый методом и тип свойства должны совпадать.

После **write** указывается имя поля или имя метода в который осуществляет запись значения свойства. Тип свойства и тип поля, в которое осуществляется запись, должны совпадать. Если свойство записывает значение с помощью метода, то метод должен иметь следующий заголовок

```
procedure <имя метода>(Value: <тип свойства>;
```

Обратите внимание на то, что тип параметра, передаваемый в процедуру и тип свойства должны совпадать.

Секция **read** или секция **write** могут отсутствовать, таким образом можно создавать свойства только для чтения или только для записи. Обращение к свойствам производится аналогично полям, т.е. с помощью оператора «.». Свойствам можно присваивать значения и использовать при вычислении выражений.

 **Совет:** Для методов использующихся для записи и чтения свойств принято применять следующую схему именованя. Для методов читающих свойство: *Get<имя свойства>*. Для методов, записывающих свойство: *Set<имя свойства>*. Располагаются методы работающие со свойствами в секции **private** или **protected**.

Рассмотрим пример

Листинг 59

```
type
  TAddClass=class
  private
    FFirstArg,
    FSecondArg:Integer;
  public
    function GetSum:Integer;
    property FirstArg:Integer read FFirstArg write FFirstArg;
    property SecondArg:Integer read FSecondArg write FSecondArg;
    property Sum:Integer read GetSum;
  end;

function TAddClass.GetSum:Integer;
begin
  Result:=FirstArg + SecondArg;
end;

//основная программа
var Add:TAddClass;
  Arg:Integer;
begin
  Add:=TAddClass.Create;

  write('First > '); //вводим первое слагаемое
  readln(Arg);
  Add.FirstArg:=Arg;

  write('Second > '); //вводим второе слагаемое
  readln(Arg);
  Add.SecondArg:=Arg

  writeln(Add.Sum); //выводим результат
  Add.Free;
  readln;
end.
```

Класс, описанный в листинге 6, реализует сложение двух целых чисел. Аргументы вводятся с помощью свойств FirstArg и SecondArg. Результат считывается из свойства Sum, доступного только для чтения.

Обратите внимание, что свойство нельзя использовать как параметр-переменную, т.е. следующий код неработоспособен

```
readln(Add.SecondArg);
```

При принятии решения о выборе поля или метода при создании свойства следует руководствоваться следующими простыми принципами.

Для записи свойства используется метод в том случае, если кроме изменения поля объекта необходимо произвести некоторые дополнительные действия, например, проверить корректность данных, вывести данные на экран, изменить другие поля объекта в соответствии с вводимыми данными и т.п. Более того, свойству может быть вообще не сопоставлено конкретное поле, а значение свойства использоваться при реализации какого-либо внутреннего алгоритма. В иных случаях для записи лучше использовать непосредственно поле. В любом случае, при необходимости, можно заменить поле на метод в описании свойства, при этом для основной программы такая модификация пройдет незаметно.

Для чтения свойства обычно применяется непосредственно поле, если свойству не сопоставлено поле, т.е. оно является вычисляемым, то используют метод (см. свойство Sum). Под вычисляемым свойством понимается такое свойство, для получения значения которого необходимо выполнить специальный код, например если значение свойства хранится в файле, то для его получения необходимо открыть файл и считать некоторое значение.



**Совет:** Для ускорения набора текста программы можно воспользоваться технологией **Code Completion** среды **Delphi**. Для этого необходимо установить курсор в строку с описанием класса и нажать **Ctrl+Shift+C**. После этого система автоматически добавит все недостающие описания и заготовки тел методов. Например, для примера приведенного выше достаточно ввести следующее описание класса

```
TAddClass=class
```

```
public
```

```
property FirstArg:Integer read FFirstArg write FFirstArg;
```

```
property SecondArg:Integer read FSecondArg write FSecondArg;
```

```
property Sum:Integer read GetSum;
```

```
end;
```

Остальные описания будут добавлены автоматически.

## Свойства массивы

Кроме создания обычных свойств возможно создание свойств ведущих себя как массивы. Синтаксис описания таких свойств следующий

```
property <имя свойства>[<список индексов>]:<тип элемента>
```

```
read <имя метода> write <имя метода>;
```

здесь <список индексов> – список индексов, причем индекс может быть любого типа включая объектный, число индексов неограниченно. Синтаксис объявления списка индексов следующий

```
<id>, <id>, ..., <id>:<тип>; <id>, <id>, ..., <id>:<тип>;...
```

В отличие от обычных свойств, свойства массивы используют только методы для чтения и записи своих значений. Ниже приведено несколько примеров объявления свойств массивов.

```
//свойство массив в виде одномерного массива  
property Items[I:integer]:real read GetItems write SetItems;
```

```
//свойство массив в виде двумерного массива  
property Items[I, J:integer]:real read GetItems write SetItems;
```

```
//свойство массив в виде одномерного массива, индекс строковый  
property Items[s:string]:real read GetItems write SetItems;
```

Синтаксис описания методов для работы со свойствами массивами запоминать не обязательно, можно воспользоваться технологией Code Completion среды Delphi. Для этого необходимо просто напечатать, например

```
property Items[I:integer]:real;
```

Остальные элементы система вставит самостоятельно.

Рассмотрим пример реализации объекта со свойством массивом в виде одномерного массива.

Листинг 60

```
type
  TMyList = class
  private
    FData:array[1..3] of string;
    function GetItems(index: Integer): string;
    procedure SetItems(index: Integer; const Value: string);
  public
    property Items[index:Integer]:string read GetItems write
    SetItems;
  end;
{ TMyList }

function TMyList.GetItems(index: Integer): string;
begin
  if (index>0)and(index<4) then Result:=FData[index]
  else Result:='no item';
end;

procedure TMyList.SetItems(index: Integer; const Value: string);
begin
  if (index>0)and(index<4) then FData[index]:=Value;
end;

var List:TMyList;
    i:integer;
begin
  List:=TMyList.Create;

  List.Items[1]:='First';
  List.Items[2]:='Second';
  List.Items[3]:='Third';

  for i:=1 to 10 do writeln(List.items[i]);

  List.Free;
  readln;
end.
```

Рассмотрим подробнее методы осуществляющие запись и чтение свойства Items. Для чтения свойства применяется метод имеющий следующий заголовок.

```
function GetItems(index: Integer): string;
```

В качестве параметра методу передается индекс элемента. Функция возвращает элемент соответствующий индексу. Для записи свойства используется следующий метод.

```
procedure SetItems(index: Integer; const Value: string);
```

В качестве первого параметра передается индекс элемента для записи. В качестве второго передается значение элемента.

## Перекрытие свойств

Свойство базового класса можно перекрыть в производном классе, например чтобы добавить ему новый атрибут доступа или связать с другим полем или методом. Для этого необходимо заново объявить свойство с новыми методами (полями) осуществляющими запись или чтение свойства в производном классе.

## Параметр Self

Кроме полей и методов в каждом объекте содержится неявный параметр Self. Этот параметр используется, если объект должен передать себя в качестве параметра (свой адрес). При вызове методов класса в качестве неявного параметра в тело метода всегда передается параметр Self, этим отличается вызов обычной подпрограммы от вызова метода. Именно по этому параметру метод определяет с данными, какого объекта он работает, ведь весь класс объектов обслуживает только один экземпляр метода.

## Контроль и преобразование типов

При присваивании переменных объектного типа принято следующее соглашение – объектной переменной родительского класса можно присвоить значение объектной переменной любого дочернего класса, но не наоборот. Поскольку реальный экземпляр объекта может оказаться наследником класса, указанного при описании объектной переменной, то возникает потребность в проверке, к какому классу принадлежит объект на самом деле. Для осуществления такой проверки служит оператор **is**. Если объект принадлежит к указанному классу, то результат вычисления оператора true, иначе false, например

```
if Obj is TMyClass then ...
```

Преобразование объектной переменной к другому объектному типу выполняется с помощью оператора **as**, например

```
MyObj := Obj as TMyClass;
```

Следует отметить, что объект действительно должен быть того типа к которому его преобразуют, т.е. выполняется именно не конвертация, как, например, перевод строки в число, а именно приведение типа. Приведение объектных типов можно также выполнить с помощью стандартного оператора приведения типа, например

```
MyObj := TMyClass(Obj);
```

## Делегирование методов

В языке **Object Pascal** существуют процедурные типы данных для методов объектов. Внешне объявление процедурного типа для метода отличается от обычного словосочетанием **of object**, записанным после прототипа процедуры или функции:

```
type  
TEvent = procedure (Sender: TObject) of object;
```

Переменная такого типа называется указателем на метод. Она занимает в памяти 8 байт и хранит одновременно ссылку на объект и адрес его метода. В переменную такого типа можно записать метод, а затем вызвать его передав ему необходимые данные.

С помощью указателей на методы организуется механизм называемый делегированием. Этот механизм позволяет передать часть работы другому объекту, например, сосредоточить в одном объекте обработку событий, возникающих в других объектах. Это избавляет программиста от необходимости порождать многочисленные классы-наследники и перекрывать в них виртуальные методы, т.е. делегирование позволяет изменять поведение отдельного объекта, а не всего класса, как это происходит при использовании механизмов раннего и позднего связывания. Делегирование широко применяется в среде Delphi. Например, все компоненты (кнопки, строки ввода, ...) делегируют обработку своих событий той форме, в которую они помещены.

## Пример класса

Ниже приведен пример создания класса. В данном примере описан базовый класс TPerson и дочерний TStudent. Продемонстрирована методика перекрытия методов базового класса на примере 3-х методов AssignTo, ShowInfo, Input.

Листинг 61 – Пример класса

```
program MyClass;
{$APPTYPE CONSOLE}
uses SysUtils, Classes;

type TPerson=class(TPersistent)
  private
    FAge: Integer;
    FName: string;
    FFamilia: string;
    procedure SetAge(const Value: Integer);
    procedure SetFamilia(const Value: string);
    procedure SetName(const Value: string);
  protected
    {унаследован от TPersistent копирует содержимое
    объекта в объект Dest}
    procedure AssignTo(Dest: TPersistent); override;
  public
    procedure ShowInfo; virtual; //показывает информацию
    procedure Input; virtual; //ВВОД ДАННЫХ
    //фамилия
    property Familia:string read FFamilia write SetFamilia;
    //имя
    property Name:string read FName write SetName;
    //возраст
    property Age:Integer read FAge write SetAge;
end;

TStudent=class(TPerson)
  private
    FGruppa: Integer;
    procedure SetGruppa(const Value: Integer);
  protected
    procedure AssignTo(Dest: TPersistent); override;
  public
    procedure ShowInfo; override;
    procedure Input; override;
    //номер учебной группы
    property Gruppa:Integer read FGruppa write SetGruppa;
end;

{ TPerson }

procedure TPerson.SetAge(const Value: Integer);
begin
  FAge := Value;
end;

procedure TPerson.SetFamilia(const Value: string);
begin
  FFamilia := Value;
end;
```

```

procedure TPerson.SetName(const Value: string);
begin
    FName := Value;
end;

procedure TPerson.ShowInfo;
begin
    writeln(Familia, ' ', Name, ' Age :', Age);
end;

procedure TPerson.Input;
begin
    write('Familia > ');
    readln(FFamilia);
    write('Name > ');
    readln(FName);
    write('Age > ');
    readln(FAge);
end;

procedure TPerson.AssignTo(Dest: TPersistent);
begin
    if (Dest is TPerson)or(Dest is TStudent) then
        begin
            TPerson(Dest).Familia:=Familia;
            TPerson(Dest).Name:=Name;
            TPerson(Dest).Age:=Age;
        end
    else inherited;
end;

{ TStudent }
procedure TStudent.AssignTo(Dest: TPersistent);
begin
    if (Dest is TStudent) then
        begin
            TStudent(Dest).Familia:=Familia;
            TStudent(Dest).Name:=Name;
            TStudent(Dest).Age:=Age;
            TStudent(Dest).Gruppa:=Gruppa;
        end
    else
        if (Dest is TPerson) then
            begin
                TPerson(Dest).Familia:=Familia;
                TPerson(Dest).Name:=Name;
                TPerson(Dest).Age:=Age;
            end
        else inherited;
    end;

procedure TStudent.Input;
begin
    inherited Input;
    write('Gruppa > ');
    readln(FGruppa);
end;

procedure TStudent.SetGruppa(const Value: Integer);

```

```

begin
  FGruppa := value;
end;

procedure TStudent.ShowInfo;
begin
  writeln(Familia, ' ', Name, ' Age :', Age, ' Gruppa :',
Gruppa);
end;

var Person:TPerson;
    Student:TStudent;
begin
  Person:=TPerson.Create;
  Person.Input;
  writeln('TPerson Info');
  Person.ShowInfo;
  writeln;
  writeln('copy from TPerson');

  Student:=TStudent.Create;
  Student.Assign(Person);
  Student.ShowInfo;

  writeln;
  Student.Input;
  writeln('TStudent Info');
  Student.ShowInfo;

  Person.Free;
  Student.Free;

  writeln('Press Enter to Exit');
  readln;
end.

```

## Некоторые стандартные классы Object Pascal

Ниже рассмотрены некоторые классы **Object Pascal**.

### **TObject**

Класс **TObject** является родительским для всей иерархии классов **Object Pascal**. Он содержит ряд методов, которые по наследству передаются всем остальным классам. Среди них конструктор **Create**, деструктор **Destroy**, метод **Free**, а также некоторые другие методы. Класс **TObject** объявлен в модуле **System**. В таблице приведены некоторые свойства и методы класса **TObject**.

Таблица 24 – Методы класса **TObject**

<i>Метод</i>	<i>Описание</i>
<b>constructor</b> Create;	Стандартный конструктор
<b>procedure</b> Free;	Уничтожает объект: вызывает стандартный деструктор <b>Destroy</b> , если значение псевдопеременной <b>Self</b> не равно <b>nil</b> .
<b>class function</b> ClassName: ShortString;	Возвращает имя класса



<i>Метод</i>	<i>Описание</i>
<b>class function</b> ClassNameIs(const Name: string): Boolean; <b>class function</b> InstanceSize: Longint;	Проверяет, является ли заданная строка именем класса  Возвращает количество байт, необходимых для хранения в памяти одного объекта соответствующего класса. Заметим, что значение, возвращаемое этим методом и значение, возвращаемое функцией SizeOf при передаче ей в качестве аргумента объектной переменной – это разные значения. Функция SizeOf всегда возвращает значение 4 (SizeOf(Pointer)), поскольку объектная переменная – это не что иное, как ссылка на данные объекта в памяти. Значение InstanceSize – это размер этих данных, а не размер объектной переменной.
<b>class function</b> NewInstance: TObject; <b>virtual</b> ;	Вызывается при создании объекта для выделения динамической памяти, чтобы разместить в ней данные объекта. Метод вызывается автоматически, поэтому нет необходимости вызывать его явно
<b>procedure</b> FreeInstance; <b>virtual</b> ;	Вызывается при уничтожении объекта для освобождения занятой объектом динамической памяти. Метод вызывается автоматически, поэтому нет необходимости вызывать его явно
<b>destructor</b> Destroy; <b>virtual</b> ;	Стандартный деструктор

Класс **TObject** выполняет лишь работу по размещению и удалению объекта в памяти, а также некоторые другие «сервисные» функции, связанные с функционированием программы. Никакие прикладные задачи не могут быть решены с помощью этого объекта.

## ***TPersistent***

Класс **TPersistent** содержит очень важные методы обеспечивающие взаимодействие объектов данного класса (а следовательно и его потомков) со средой разработки.

### **procedure** Assign(Source: TPersistent);

Этот метод осуществляет копирование содержимого объекта Source в текущий объект. Именно с помощью этого метода создаются копии объектов, т.к. как уже говорилось выше переменные объектного типа, по сути, являются указателями. Для корректного копирования данных объекта необходимо перекрывать этот метод во всех потомках класса **TPersistent**. Более корректным будет перекрытие не метода Assign, а метода AssignTo, находящегося в защищенной секции класса **TPersistent**. Так как, этот метод вызывается при использовании метода Assign. Пример создания копии объекта.

```
var p1, p2: TMyObject;
...
p1:=TMyObject.Create;
p2:=TMyObject.Create;
...
p1:=p2; //неправильно
p1.Assign(p2); //правильно
```

## **TList**

Для использования класса **TList** необходимо подключить к программе модуль **Classes**. Класс **TList** позволяет создать набор из произвольного количества элементов и организовать индексный способ доступа к ним, как это делается при работе с массивом. В отличие от массивов список имеет две важные особенности: способность динамически изменять свой размер в ходе работы программы и способность хранить элементы разных типов.

Фактически списки представляют собой массивы нетипизированных указателей на размещенные в динамической памяти элементы, сами массивы также размещаются в динамической памяти. Некоторые свойства класса **TList** приведены в таблице 1.

Таблица 25 – Свойства класса **TList**

<i>Свойство</i>	<i>Описание</i>
<b>property</b> Capacity: Integer;	Содержит количество элементов массива указателей коллекции; всегда больше Count. Если при добавлении очередного элемента Count стало равно Capacity, происходит автоматическое расширение списка на 16 элементов.
<b>property</b> Count: Integer;	Количество элементов списка. Это свойство изменяется при добавлении или удалении элемента.
<b>property</b> Items(Index: Integer): Pointer;	Возвращает указатель на элемент списка по его индексу. Самый первый элемент списка имеет индекс 0.

Обратите внимание, что свойство **Count** определяет количество помещенных в список элементов, а свойство **Capacity** текущую емкость списка. Если при добавлении очередного элемента обнаруживается, что емкость списка исчерпана, происходит наращивание емкости на фиксированную величину. Если заранее известно, сколько элементов необходимо поместить в список, то для более эффективной работы программы лучше заранее установить нужное значение в свойство **Capacity**. Для работы со списком определено достаточно много методов некоторые из них приведены в таблице 2.

Таблица 26 – Методы класса **TList**

<i>Методы</i>	<i>Описание</i>
<b>function</b> Add(Item: Pointer): Integer;	Добавляет элемент <b>Item</b> в конец списка и возвращает его индекс.
<b>procedure</b> Clear;	Очищает список, удаляя из него все элементы.
<b>procedure</b> Delete(Index: Integer) ;	Удаляет из списка элемент с индексом <b>Index</b> .
<b>procedure</b> Exchange(Index1, Index2: Integer);	Меняет местами элементы с индексами <b>Index1</b> и <b>Index2</b> .
<b>function</b> Expand: TList;	Расширяет массив элементов, увеличивая <b>Capacity</b> .
<b>function</b> First: Pointer;	Возвращает указатель на самый первый элемент списка.
<b>function</b> IndexOf(Item: Pointer): Integer;	Отыскивает в списке элемент <b>Item</b> и возвращает его индекс.

<i>Методы</i>	<i>Описание</i>
<b>procedure</b> Insert(Index: Integer; Item: Pointer);	Вставляет элемент Item в позицию Index списка: новый элемент получает индекс Index, все элементы с индексами Index и больше увеличивают свой индекс на 1. При необходимости расширяет список.
<b>function</b> Last: Pointer;	Возвращает указатель на последний элемент списка.
<b>procedure</b> Move(CurIndex, NewIndex: Integer);	Перемещает элемент в списке с позиции CurIndex в позицию NewIndex. Все элементы старого списка с индексами от CurIndex до NewIndex уменьшают свой индекс на 1.
<b>procedure</b> Pack;	Упаковывает список: удаляет пустые элементы в конце массива индексов.
<b>function</b> Remove(Item: Pointer): Integer;	Отыскивает в списке элемент Item и удаляет его.
<b>procedure</b> Sort(Compare: TListSortCompare);	Сортирует список с помощью функции Compare.

Следует помнить о следующей особенности работы методов Add и Insert класса TList. В качестве параметров эти методы получают указатель на вставляемый элемент, но за размещение данных размещенных по указателю отвечает программист, т.е. необходимо самостоятельно создать динамическую переменную и получить соответствующий указатель. Точно так же методы Delete, Remove и Clear не уничтожают распределенные в памяти данные, которые программист должен, если это необходимо, уничтожить сам.

Метод Sort сортирует список по критерию, устанавливаемому функцией Compare. Тип TListSortCompare определен следующим образом:

TListSortCompare = **function**(Item1, Item2: Pointer): Integer;

В качестве параметров, функция Compare получает два указателя на сравниваемые элементы списка. Функция должна возвращать результат по ниже приведенным правилам:

- любое отрицательное число, если  $Item1 < Item2$
- 0, если  $Item1 = Item2$ ;
- любое положительное число, если  $Item1 > Item2$ .

В следующем примере в список List помещается 20 случайных вещественных чисел, равномерно распределенных в диапазоне 0...1. Список сортируется по возрастанию чисел и отображается экране.

**Листинг 62 – Работа с объектом TList**

```

program DemoList;
{$APPTYPE CONSOLE}
uses SysUtils, Classes;

function CompareFunc(Item1, Item2: Pointer): Integer;
begin
  if Integer(Item1^)>Integer(Item2^) then result:=1
  else result:=-1;
end;

```

```

const n=10; // число элементов в списке
var List:TList; // список
    pInt:^Integer;
    i:Integer;
begin
    //создаем список
    List:=TList.Create;

    //заполняем список элементами
    for i:=1 to n do
        begin
            new(pInt);
            pInt^:=random(100);
            List.Add(pInt);
        end;

    //вывод списка на экран
    for i:=0 to List.Count-1 do
        write(Integer(List.Items[i]^), ' ');

    writeln;

    //сортировка списка
    List.Sort(CompareFunc);

    writeln('Sort List');
    for i:=0 to List.Count-1 do
        write(Integer(List.Items[i]^), ' ');

    writeln;

    //уничтожаем список
    //освобождаем память занятую каждым из объектов
    for i:=0 to List.Count-1 do
        begin
            Dispose(List.Items[i]);
            List.Items[i]:=nil;
        end;

    //удаляем объект список
    List.Free;

    writeln('Press Enter to exit');
    readln;
end.

```

## ***TStrings u TStringList***

Абстрактный класс **TStrings**<sup>10</sup> инкапсулирует поля и методы для работы с наборами строк. Замечательной особенностью **TStrings** и его потомков является то обстоятельство, что элементами наборов служат пары строка-объект, в которых строка – собственно строка символов, а объект – объект любого класса Delphi. Такая двойственность позволяет сохранять в **TStrings** объекты с текстовыми примечаниями, сортировать объекты, отыскивать нужный объект по его описанию и, т.д. Кроме того, в качестве объекта может использоваться потомок от **TStrings**, что позволяет создавать многомерные наборы строк. Так как класс **TStrings** является абстрактным, то его непосредственное использование в

<sup>10</sup> Описан этот класс в модуле Classes

программах недопустимо. Поэтому рассмотрим один из многочисленных дочерних классов образованных от **TStrings** – класс **TStringList**.

Класс **TStringList** содержит средства обслуживания строк вида Name = Value, где Name – имя параметра, Value – его значение. Такие строки обычно используются в файлах настройки (ini Файлы).

Набор строк реализуется подобно **TList** – в виде массива указателей (на строки и объекты). Свойство Capacity показывает текущую длину этого массива, а свойство Count – количество элементов, занятых в нем. Некоторые свойства класса **TStringList** приведены в таблице 3.

Таблица 27 – Свойства класса **TStringList**

<i>Свойство</i>	<i>Описание</i>
<b>property</b> Capacity: Integer;	Текущая емкость набора строк.
<b>property</b> CommaText: String;	Служит для установки или получения всего набора строк в виде единой строки с кавычками и запятыми.
<b>property</b> Count: Integer;	Текущее количество строк в наборе.
<b>property</b> Duplicates: TDuplicates;	Свойство, позволяющее управлять возможностью размещения в наборе двух и более идентичных строк.
<b>property</b> Names[Index: Integer]: String;	Для строки с индексом Index возвращает часть Name. если это строка вида Name=Value, в противном случае возвращает пустую строку.
<b>property</b> Objects[Index: Integer]: TObject;	Открывает доступ к объекту, связанному со строкой с индексом Index.
<b>property</b> Sorted: Boolean;	Признак необходимости сортировки строк в алфавитном порядке.
<b>property</b> Strings[Index: Integer]: String;	Открывает доступ к строке с индексом Index.
<b>property</b> Text: String,-	Интерпретирует набор строк в виде одной длинной строки с внутренними разделителями EOLN между отдельными строками набора.
<b>property</b> Values[const Name: String]: String;	По части Name отыскивает в наборе и возвращает часть Value для строк вида Name=Value.

Свойство CommaText интерпретирует содержимое набора строк в виде одной длинной строки с элементами вида "Первая строка", "Вторая строка", "Третья строка" и т.д. (каждая строка набора заключается в двойные кавычки и отделяется от соседней строки запятой; если в строке встречается символ «"», он удваивается). Свойство Text интерпретирует содержимое набора в виде одной длинной строки с элементами, разделенными стандартным признаком EOLN (# 13 # 10).

При Sorted = False строки набора автоматически сортируются в алфавитном порядке. При этом свойство Duplicates разрешает коллизию, связанную с добавлением в набор строки, идентичной одной из ранее вставленных. Если Duplicates = duIgnore, идентичная строка отвергается и программе ничего об этом не сообщается; если Duplicates = duError, возбуждается исключение EListError, значение Duplicates = duAccept разрешает вставлять в набор сколько угодно идентичных строк.

В таблице 4 приведены некоторые методы класса TS  
Таблица 28 – Методы класса **TStringList**

--	--

<i>Метод</i>	<i>Описание</i>
<b>function</b> Add(const S: String): Integer;	Добавляет строку в набор данных и возвращает ее индекс.
<b>function</b> AddObject(const S: String; AObject: TObject): Integer;	Добавляет строку и объект в набор данных.
<b>procedure</b> AddStrings(Strings: TStrings);	Добавляет к текущему набору новый набор строк.
<b>procedure</b> Assign(Source: TPersistent);	Уничтожает прежний набор строк и загружает из Source новый набор. В случае неудачи возникает исключение EConvertError.
<b>procedure</b> Clear;	Очищает набор данных и освобождает связанную с ним память.
<b>procedure</b> Delete(Index: Integer) ;	Уничтожает элемент набора с индексом Index и освобождает связанную с ним память.
<b>function</b> Equals(Strings: TStrings): Boolean;	Сравнивает построчно текущий набор данных с набором Strings и возвращает True, если наборы идентичны.
<b>function</b> Find(const S: String; var Index: Integer): Boolean;	Ищет в наборе строку S и в случае успеха в параметре Index возвращает ее индекс.
<b>function</b> IndexOf(const S: String): Integer;	Для строки S возвращает ее индекс или -1, если такой строки в наборе нет.
<b>function</b> IndexOfObject(AObject: TObject): Integer;	Для объекта AObject возвращает индекс строки или -1, если такого объекта в наборе нет.
<b>procedure</b> Insert(Index: Integer; const S: String);	Вставляет строку в набор и присваивает ей индекс Index
<b>procedure</b> InsertObject(Index: Integer; const S: String; AObject: TObject)	Вставляет строку и объект в набор и присваивает им индекс Index.
<b>procedure</b> LoadFromFile(const FileName: String);	Загружает набор из файла.
<b>procedure</b> LoadFromStream(Stream: TStream);	Загружает набор из потока.
<b>procedure</b> SaveToFile(const FileName: String);	Сохраняет набор в файле.
<b>procedure</b> SaveToStream(Stream: TStream);	Сохраняет набор в потоке.

Ниже приведен простой пример чтения содержимого текстового файла и вывод его содержимого на экран.

Листинг 63 – Просмотр содержимого текстового файла

```

program ViewTxt;
{$APPTYPE CONSOLE}
uses Classes;
var List:TStringList;

```

```

    FileName:string;
    i:Integer;
begin
write('filename > ');
readln(FileName);

List:=TStringList.Create;
List.LoadFromFile(FileName);

for i:=0 to List.Count-1 do writeln(List.Strings[i]);

List.Free;
writeln('Press Enter to Exit');
readln;
end.

```

### ***Потоковый ввод-вывод класс TStream***

Под потоком данных понимается некоторое абстрактное устройство, которое может записывать и считывать данные, а также смещать текущую позицию в данных.

Класс **TStream** является базовым классом для потоков разных типов. В этом классе реализованы все необходимые свойства и методы, необходимые для чтения и записи данных на различные типы носителей (память, диск, медиа носители). Благодаря этому классу, доступ к разным типам носителей становится одинаковым. Класс **TStream** является абстрактным и поэтому объекты этого класса никогда напрямую не используются. Используются потомки этого класса, например, класс **TFileStream** является потомком базового класса **TStream** и позволяет получить доступ к диску. Точно так же можно получить доступ:

- к памяти через класс **TMemoryStream**.
- к сети через класс **TWinSocketStream**.
- к СОМ интерфейсу через **TOleStream**.
- к строкам, находящимся в динамической памяти **TStringStream**.

Класс TStream описан в модуле Classes, там же описаны многие его потомки. На рисунке 7 приведена иерархия классов потоков.

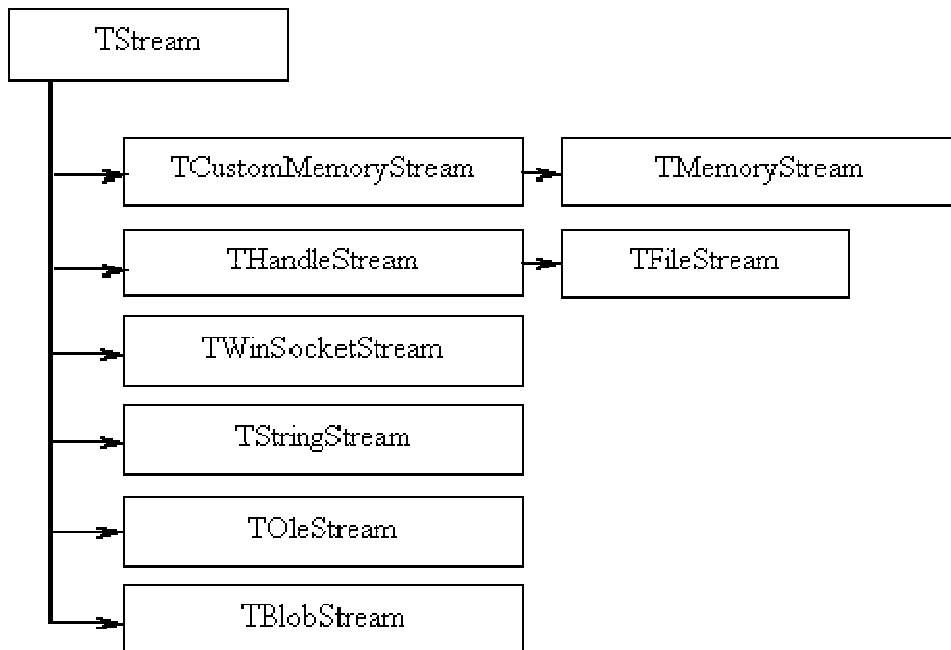


Рисунок 25 – Иерархия классов потоков

Это не полный список классов потоков, но даже все эти класс мы рассматривать не будем. Рассмотрим только базовый класс **TStream**, а также пример работы с файловым потоком **TFileStream**.

Ниже приведены некоторые свойства и методы класса TStream.

Таблица 29 – Свойства объекта TStream

<i>Свойства</i>	<i>Описание</i>
<b>property</b> Position: Int64;	указывает на текущую позицию курсора в потоке. Начиная с этой позиции будет происходить чтение данных.
<b>property</b> Size: Int64;	размер данных в потоке.

Таблица 30 – Методы класса TStream

<i>Методы</i>	<i>Описание</i>
<b>function</b> CopyFrom(Source: TStream; Count: Int64): Int64;	Метод копирует данные из другого потока. Source – источник данных. Count – размер копируемых данных в байтах.
<b>function</b> Read(var Buffer; Count: Longint): Longint;	Считывание данных из потока, начиная с текущей позиции курсора. Buffer – буфер для данных. Count – размер считываемых данных.
<b>function</b> Write(const Buffer; Count: Longint): Longint;	Записывает данные в поток, начиная с текущей позиции. Buffer – буфер содержащий данные для записи. Count – размер записываемых данных.



<i>Методы</i>	<i>Описание</i>
<b>function</b> Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;	Перемещение в новую позицию в потоке. Offset – число байт на которое нужно изменить позицию. При этом отрицательное смещение означает смещение в обратном направлении. Origin – откуда надо двигаться, возможны три варианта: <ul style="list-style-type: none"> <li>- soFromBeginning – двигаться на указанные количество байт от начала файла.</li> <li>- soFromCurrent - двигаться на указанные количество байт от текущей позиции в файле к концу файла.</li> <li>- soFromEnd – двигаться от конца файла к началу на указанное количество байт.</li> </ul>
<b>procedure</b> SetSize(const NewSize: Int64);	Устанавливает размер потока. NewSize – число, новый размер потока. Например при использовании потока TFileStream можно уменьшить или увеличить размер файла.

## TFileStream

Класс **TFileStream** предназначен для работы с файлами. По сути этот класс по своим функциям аналогичен нетипизированным файлам **Object Pascal**. Являясь наследником **TStream** он наследует все его свойства и методы, а также переопределяет некоторые из них. Для создания экземпляра класса **TFileStream** удобно использовать следующую форму конструктора.

**constructor** Create(const FileName: string; Mode: Word);

где FileName – имя файла в котором находятся данные потока;

Mode – режим работы с потоком, см. таблицу 8.

Таблица 31 – Режимы работы потока

<i>Режим</i>	<i>Описание</i>
fmCreate	Создание файла с заданным именем, если файл с таким именем уже существует, то он открывается в режиме для записи.
fmOpenRead	Открывает файл в режиме только для чтения, если файл не существует, то происходит ошибка.
fmOpenWrite	Открывает файл в режиме только для записи. Запись в файл происходит поверх имеющихся в нем данных.
fmOpenReadWrite	Открывает файл в режиме как для записи, так и для чтения.

Ниже приведен пример программы для копирования файла из лабораторной работы № 6, но с использованием потоков.

Листинг 64 – Копирование файлов

```
program CopyFile;
{$APPTYPE CONSOLE}
uses SysUtils, Classes;
```

```

var Srm1, //ВХОДНОЙ ПОТОК
    Srm2:TFileStream; //ВЫХОДНОЙ ПОТОК
    SourceFileName, //ИМЯ ВХОДНОГО ФАЙЛА
    DestFileName:string; //ИМЯ ФАЙЛА КОПИИ
begin
    //открываем файл источник данных
    write('Source file >> ');
    readln(SourceFileName);
    //создаем поток в режиме только для чтения
    Srm1:=TFileStream.Create(SourceFileName, fmOpenRead);

    //создаем файл приемник
    write('Destination file >> ');
    readln(DestFileName);
    //создаем поток в режиме создания нового потока
    Srm2:=TFileStream.Create(DestFileName, fmCreate);

    //копирование содержимого потока Srm1 в Srm2
    Srm2.CopyFrom(Srm1, Srm1.Size);

    //уничтожение объектов
    Srm1.Free;
    Srm2.Free;

    writeln('File success copy');
    writeln('Press Enter to exit');
    readln;
end.

```

Ниже приведена программа для чтения информации из mp3 файла. Эти файлы часто содержат информацию о звуковых данных хранящихся в файле. Эти данные хранятся в специальном блоке данных размеров в 128 байт, который имеет название ID3 tag. Если этот блок присутствует в файле, то он располагается в конце файла. Начинается этот блок со специальной комбинации символов, а именно «TAG». Таким образом, можно определить имеется ли в файле этот специальный информационный блок данных. Общая структура этого блока приведена на рисунке



a)



б)

Рисунок 26 – Общая структура ID3 tag. а) Версия 1.0. б) Версия 1.1

Программа реализующая чтение информации из mp3-файла приведена ниже.  
 Листинг 65 – Чтение ID3 tag из mp3 файла

```

program mp3Tag;

{$APPTYPE CONSOLE}

uses
    SysUtils,
    Classes,
    mp3_list in 'mp3_list.pas';

var Stream:TFileStream; //файловый поток
    FileName:String; //имя файла
    buf :byte;
    Tag_Type:(id3v1, id3v11); //версия заголовка

{Считывает массив символов из потока и преобразует его в строку
 Size - число символов в строке}
function GetString(Size:Integer):string;
var Buf:PChar;
begin
    GetMem(Buf, Size); //память под временный буфер
    Stream.Read(Buf^, Size); //считываем данные в буфер
    SetString(Result, Buf, Size); //копируем данные в строку
    FreeMem(Buf); //освобождаем буфер
end;

begin
    //ввод имени файла
    write('filename > ');
    readln(FileName);

    //создаем поток
    Stream:=TFileStream.Create(FileName, fmOpenRead);

    //позиционируем указатель потока на 128 байт от конца файла
    Stream.Seek(-128, soFromEnd);

    //считываем идентификатор ID3 tag
    if GetString(3)='TAG' then //если файл содержит ID3 tag
        begin
            //определяем версию ID3 tag

            //считываем значение 3-го байта от конца файла
            Stream.Seek(-3, soFromEnd);
            Stream.Read(buf, 1);
            if buf=0 then
                begin
                    Stream.Read(buf, 1);
                    if buf<>0 then //ID3V1.1
                        Tag_Type:=id3v11
                    else Tag_Type:=id3v1;
                end
            else Tag_Type:=id3v1;

            if Tag_Type=id3v1 then
                writeln('Version ID3 V1')
            else
    
```

```

writeln('Version ID3 V1.1');

//устанавливаем указатель потока на первое поле заголовка
Stream.Seek(-125, soFromEnd);

//считываем остальные поля
writeln('Song title - ', GetString(30));
writeln('Artist     - ', GetString(30));
writeln('Album      - ', GetString(30));
writeln('Year       - ', GetString(4));

if Tag_Type=id3v1 then
  writeln('Comment   - ', GetString(30))
else
  begin
    writeln('Comment   - ', GetString(28));
    Stream.Read(buf, 1); //пропускаем 1 байт
    Stream.Read(buf, 1);
    writeln('Track - ', buf);
  end;

  Stream.Read(buf, 1);
  if buf<=High(mp3_genre) then
    writeln('Genre - ', mp3_genre[buf])
  else writeln('Genre - Unknown')
end
else
  writeln('ID3 tag not Found!');

Stream.Free; //уничтожение потока
writeln('Press Enter to Exit');
readln;
end.

```

## Задания к лабораторной работе

1. Наберите, отладьте и изучите алгоритм программ приведенных в листингах 8-12.
2. Измените программу в листинге 10 таким образом, чтобы на экран выводилось отсортированное содержимое текстового файла.
3. Напишите программу, хранящую в списке (TList) объекты TPerson и TStudent. В программе должно быть реализовано меню. Программа должна обрабатывать следующие команды:
  - добавить объект класса TPerson в список;
  - добавить объект класса TStudent в список;
  - вывести содержимое списка на экран.

## Вопросы к лабораторной работе

1. В чем заключается структурный подход к программированию?
2. В чем заключается модульный подход к программированию?
3. В чем заключается объектно-ориентированный подход к программированию?
4. Что входит в понятие класс?
5. В чем отличие понятий класс и объект?

6. В чем заключается концепция наследования?
7. В чем заключается концепция инкапсуляции?
8. В чем заключается концепция полиморфизма?
9. В чем отличие механизмов раннего и позднего связывания методов. Когда применяется механизм позднего, а когда раннего связывания?
10. В чем заключается механизм делегирования методов?
11. Какие методы называют виртуальными?
12. Как объявляется класс в Object Pascal?
13. Какие элементы содержит класс Object Pascal?
14. Каково назначение секций public, protected и private в описании класса?
15. Как объявляется метод класса?
16. Как перекрыть метод класса?
17. Как объявить виртуальный метод?
18. В чем суть механизма позднего связывания, как он реализован в Object Pascal?
19. Для чего предназначен конструктор класса, в чем его отличие от обычного метода?
20. Для чего предназначен деструктор, в чем его отличие от обычного метода?
21. Для чего предназначено зарезервированное слово **inherited**?
22. В чем заключается отличие «методов класса» от обычных методов, как описываются такие методы в Object Pascal?
23. Что такое абстрактный метод, класс? Как такие методы описываются в Object Pascal?
24. В чем заключается механизм перегрузки методов, как он реализован в Object Pascal?
25. Для чего предназначены свойства? Как описываются свойства в Object Pascal?
26. Каково назначение свойств-массивов? Какие существуют особенности при описании таких свойств?
27. В чем заключается механизм делегирования методов?
28. Для чего служит параметр Self?
29. Какие операторы служат для преобразования типа объектов?
30. С помощью какого оператора можно проверить принадлежит ли объект к заданному классу?

## Справочные таблицы

Таблица 1 – Методы класса TObject .....	152
Таблица 2 – Свойства класса TList .....	154
Таблица 3 – Методы класса TList .....	154
Таблица 4 – Свойства класса TStringList .....	157
Таблица 5 – Методы класса TStringList.....	157
Таблица 6 – Свойства объекта TStream.....	160
Таблица 7 – Методы класса TStream .....	160
Таблица 8 – Режимы работы потока.....	161

# Лабораторная работа №12

## Потоковый ввод-вывод. Класс TStream

### Введение

В данной лабораторной работе рассмотрен потоковый ввод-вывод в Object Pascal с помощью класса TStream и его потомков. Приведены примеры программ.

### Класс TStream

Под потоком данных понимается некоторое абстрактное устройство, которое может записывать и считывать данные, а также смещать текущую позицию в данных.

Класс **TStream** является базовым классом для потоков разных типов. В этом классе реализованы все необходимые свойства и методы, необходимые для чтения и записи данных на различные типы носителей (память, диск, медиа носители). Благодаря этому классу, доступ к разным типам носителей становится одинаковым. Класс **TStream** является абстрактным и поэтому объекты этого класса никогда напрямую не используются. Используются потомки этого класса, например, класс **TFileStream** является потомком базового класса **TStream** и позволяет получить доступ к диску. Точно так же можно получить доступ:

- к памяти через класс **TMemoryStream**.
- к сети через класс **TWinSocketStream**.
- к COM интерфейсу через **TOleStream**.
- к строкам, находящимся в динамической памяти **TStringStream**.

Класс TStream описан в модуле Classes, там же описаны многие его потомки. На рисунке 7 приведена иерархия классов потоков.

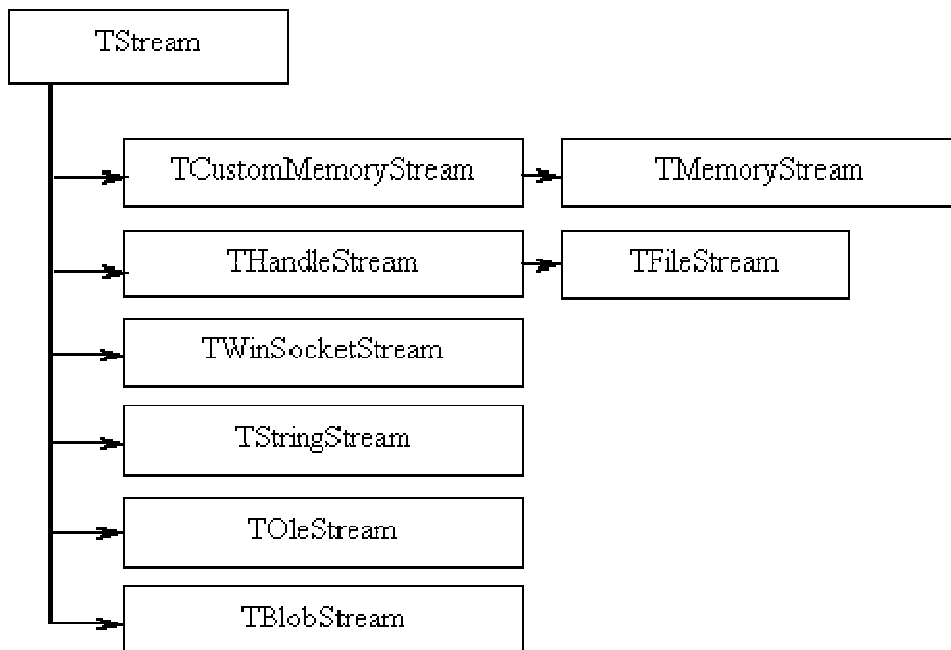


Рисунок 27 – Иерархия классов потоков

Это не полный список классов потоков, но даже все эти классы мы рассматривать не будем. Рассмотрим только базовый класс **TStream**, а также пример работы с файловым потоком **TFileStream**.

Ниже приведены некоторые свойства и методы класса TStream.

Таблица 32 – Свойства объекта TStream

<i>Свойства</i>	<i>Описание</i>
<b>property</b> Position: Int64;	указывает на текущую позицию курсора в потоке. Начиная с этой позиции будет происходить чтение данных.
<b>property</b> Size: Int64;	размер данных в потоке.

Таблица 33 – Методы класса TStream

<i>Методы</i>	<i>Описание</i>
<b>function</b> CopyFrom(Source: TStream; Count: Int64): Int64;	Метод копирует данные из другого потока. Source – источник данных. Count – размер копируемых данных в байтах.
<b>function</b> Read(var Buffer; Count: Longint): Longint;	Считывание данных из потока, начиная с текущей позиции курсора. Buffer – буфер для данных. Count – размер считываемых данных.
<b>function</b> Write(const Buffer; Count: Longint): Longint;	Записывает данные в поток, начиная с текущей позиции. Buffer – буфер содержащий данные для записи. Count – размер записываемых данных.
<b>function</b> Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;	Перемещение в новую позицию в потоке. Offset – число байт на которое нужно изменить позицию. При этом отрицательное смещение означает смещение в обратном направлении. Origin – откуда надо двигаться, возможны три варианта: <ul style="list-style-type: none"> <li>- soFromBeginning – двигаться на указанные количество байт от начала файла.</li> <li>- soFromCurrent - двигаться на указанные количество байт от текущей позиции в файле к концу файла.</li> <li>- soFromEnd – двигаться от конца файла к началу на указанное количество байт.</li> </ul>
<b>procedure</b> SetSize(const NewSize: Int64);	Устанавливает размер потока. NewSize – число, новый размер потока. Например при использовании потока TFileStream можно уменьшить или увеличить размер файла.

## TFileStream

Класс **TFileStream** предназначен для работы с файлами. По сути этот класс по своим функциям аналогичен нетипизированному файлу **Object Pascal**. Являясь наследником **TStream** он наследует все его свойства и методы, а также переопределяет некоторые из них. Для создания экземпляра класса **TFileStream** удобно использовать следующую форму конструктора.

**constructor** Create(const FileName: string; Mode: Word);

где FileName – имя файла в котором находятся данные потока;

Mode – режим работы с потоком, см. таблицу 8.

Таблица 34 – Режимы работы потока

<i>Режим</i>	<i>Описание</i>
--------------	-----------------

<i>Режим</i>	<i>Описание</i>
fmCreate	Создание файла с заданным именем, если файл таким именем уже существует, то он открывается в режиме для записи.
fmOpenRead	Открывает файл в режиме только для чтения, если файл не существует, то происходит ошибка.
fmOpenWrite	Открывает файл в режиме только для записи. Запись в файл происходит поверх имеющихся в нем данных.
fmOpenReadWrite	Открывает файл в режиме как для записи, так и для чтения.

Ниже приведен пример программы для копирования файла из лабораторной работы № 6, но с использованием потоков.

**Листинг 66 – Копирование файлов**

```

program CopyFile;

{$APPTYPE CONSOLE}

uses SysUtils, Classes;

var Srm1,           //ВХОДНОЙ ПОТОК
    Srm2:TFileStream; //ВЫХОДНОЙ ПОТОК
    SourceFileName, //ИМЯ ВХОДНОГО ФАЙЛА
    DestFileName:string; //ИМЯ ФАЙЛА КОПИИ
begin
    //открываем файл источник данных
    write('Source file >> ');
    readln(SourceFileName);
    //создаем поток в режиме только для чтения
    Srm1:=TFileStream.Create(SourceFileName, fmOpenRead);

    //создаем файл приемник
    write('Destination file >> ');
    readln(DestFileName);
    //создаем поток в режиме создания нового потока
    Srm2:=TFileStream.Create(DestFileName, fmCreate);

    //копирование содержимого потока Srm1 в Srm2
    Srm2.CopyFrom(Srm1, Srm1.Size);

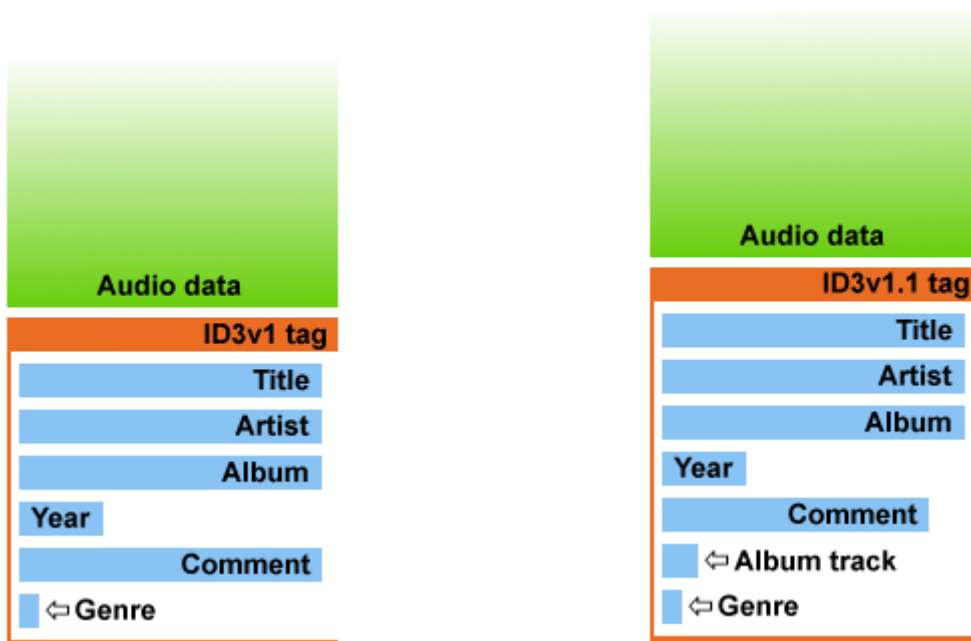
    //уничтожение объектов
    Srm1.Free;
    Srm2.Free;

    writeln('File success copy');
    writeln('Press Enter to exit');
    readln;
end.

```

Ниже приведена программа для чтения информации из mp3 файла. Эти файлы часто содержат информацию о звуковых данных хранящихся в файле. Эти данные хранятся в специальном блоке данных размеров в 128 байт, который имеет название ID3 tag. Если этот блок присутствует в файле, то он располагается в конце файла. Начинается этот блок со специальной комбинации символов, а именно «TAG». Таким образом, можно определить имеется ли в файле этот специальный информационный блок данных. Общая структура этого блока приведена на рисунке





а) б)  
Рисунок 28 – Общая структура ID3 tag. а) Версия 1.0. б) Версия 1.1

Программа реализующая чтение информации из mp3-файла приведена ниже.

Листинг 67 – Чтение ID3 tag из mp3 файла

```

program mp3Tag;
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Classes,
  mp3_list in 'mp3_list.pas';

var Stream:TFileStream; //файловый поток
      FileName:String; //имя файла
      buf :byte;
      Tag_Type:(id3v1, id3v11); //версия заголовка

{Считывает массив символов из потока и преобразует его в строку
 Size - число символов в строке}
function GetString(Size:Integer):string;
var Buf:PChar;
begin
  GetMem(Buf, Size); //память под временный буфер
  Stream.Read(Buf^, Size); //считываем данные в буфер
  SetString(Result, Buf, Size); //копируем данные в строку
  FreeMem(Buf); //освобождаем буфер
end;

begin
  //ввод имени файла
  write('filename > ');
  readln(FileName);

  //создаем поток
  Stream:=TFileStream.Create(FileName, fmOpenRead);

```

```

//позиционируем указатель потока на 128 байт от конца файла
Stream.Seek(-128, soFromEnd);

//считываем идентификатор ID3 tag
if GetString(3)='TAG' then //если файл содержит ID3 tag
begin
  //определяем версию ID3 tag

  //считываем значение 3-го байта от конца файла
  Stream.Seek(-3, soFromEnd);
  Stream.Read(buf, 1);
  if buf=0 then
  begin
    Stream.Read(buf, 1);
    if buf<>0 then //ID3v1.1
      Tag_Type:=id3v11
    else Tag_Type:=id3v1;
  end
  else Tag_Type:=id3v1;

  if Tag_Type=id3v1 then
    writeln('Version ID3 v1')
  else
    writeln('Version ID3 v1.1');

  //устанавливаем указатель потока на первое поле заголовка
  Stream.Seek(-125, soFromEnd);

  //считываем остальные поля
  writeln('Song title - ', GetString(30));
  writeln('Artist - ', GetString(30));
  writeln('Album - ', GetString(30));
  writeln('Year - ', GetString(4));

  if Tag_Type=id3v1 then
    writeln('Comment - ', GetString(30))
  else
  begin
    writeln('Comment - ', GetString(28));
    Stream.Read(buf, 1); //пропускаем 1 байт
    Stream.Read(buf, 1);
    writeln('Track - ', buf);
  end;

  Stream.Read(buf, 1);
  if buf<=High(mp3_genre) then
    writeln('Genre - ', mp3_genre[buf])
  else writeln('Genre - Unknown')
  end
else
  writeln('ID3 tag not Found!');

Stream.Free; //уничтожение потока
writeln('Press Enter to Exit');
readln;
end.

```

## Задания к лабораторной работе

4. Наберите, отладьте и изучите алгоритм программ приведенных в листингах 1 и 2.

## **Вопросы к лабораторной работе**

31. Какие классы используются для потокового ввода-вывода?
32. Какой класс служит для потокового доступа к файлам?

## **Справочные таблицы**

Таблица 1 – Свойства объекта TStream.....	168
Таблица 2 – Методы класса TStream .....	168
Таблица 3 – Режимы работы потока.....	168

# Лабораторная работа №13

## Структурированная обработка ошибок. Защищенные блоки. Классы исключений.

### Введение

В лабораторной работе рассмотрен механизм обработки исключительных ситуаций, синтаксис описания защищенных блоков. Приведены примеры программ использующих защищенные блоки.

### Механизм обработки исключительных ситуаций

#### *Защищенные блоки*

Для обработки исключительных ситуаций в **Object Pascal** предусмотрен механизм защищенных блоков. Защищенный блок начинается зарезервированным словом **try** (попытаться выполнить) и завершается словом **end**. В **Object Pascal** существует два вида защищенных блоков: блок **except** (исключить) и блок **finally** (завершить). Каждый из видов защищенных блоков подробно рассматривается ниже.

#### *Блок завершения*

Синтаксис этого защищенного блока имеет следующий вид

**try**

<операторы>

**finally**

<операторы>

**end;**

Если при выполнении операторов в секции **try finally** не произошло ошибок, то далее выполняются операторы секции **finally end**. Если при выполнении операторов в секции **try finally** в каком-либо из операторов произошла ошибка, то пропускаются все операторы начиная с ошибочного и начинают выполняться операторы секции **finally end**. Иными словами, операторы в секции finally end выполняются всегда (рисунок 1).

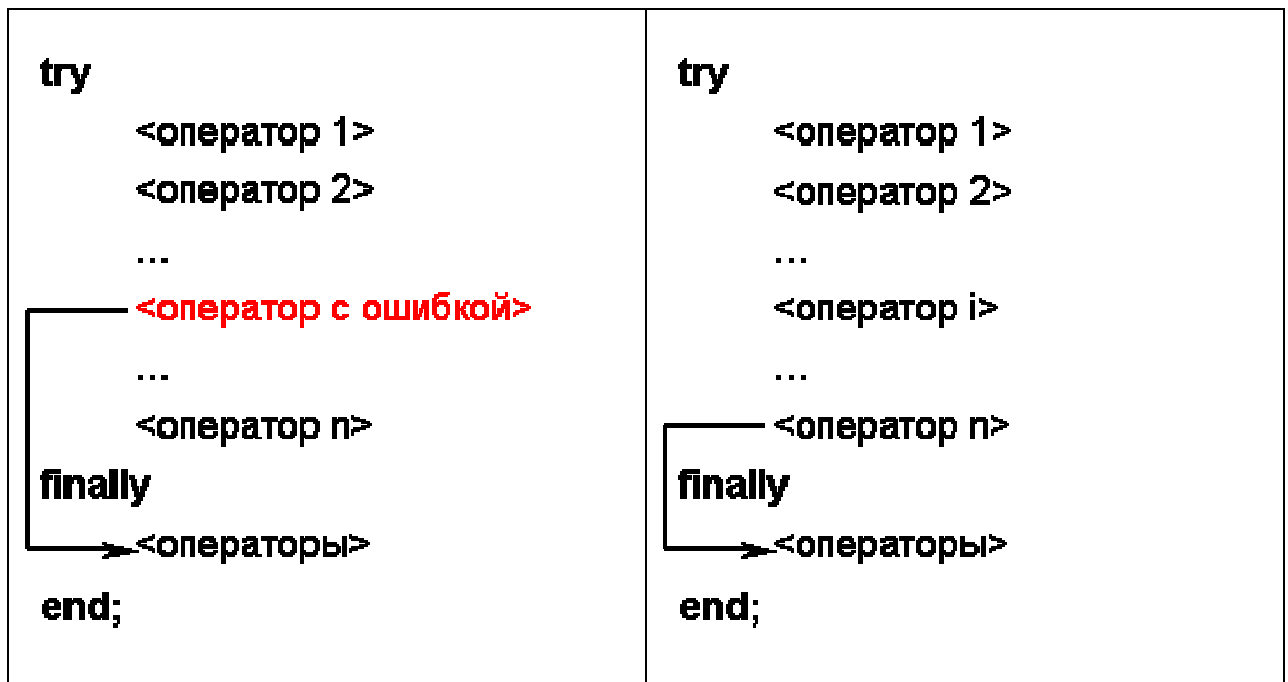


Рисунок 29 – Логика работы блока завершения

Защищенные блоки этого вида применяют для корректного освобождения ресурсов. Например, если при работе с файлом произошла какая-либо ошибка, то его необходимо закрыть. Если ошибок не было, то при завершении работы файл все равно надо закрыть. Модернизируем программу осуществляющую копирование файлов приведенную в лабораторной работе № 11, добавим обработку ошибок.

Листинг 68 – Копирование файлов

```

program CopyFile;
{$APPTYPE CONSOLE}

uses SysUtils, Classes;

var Srm1,           //входной поток
     Srm2:TFileStream; //выходной поток
     SourceFileName, //имя входного файла
     DestFileName:string; //имя файла копии
begin
    //открываем файл источник данных
    write('Source file >> ');
    readln(SourceFileName);

    try //в этом блоке может произойти ошибка

        //создаем поток в режиме только для чтения
        Srm1:=TFileStream.Create(SourceFileName, fmOpenRead);

        //создаем файл приемник
        write('Destination file >> ');
        readln(DestFileName);
        //создаем поток в режиме создания нового потока
        Srm2:=TFileStream.Create(DestFileName, fmCreate);

        //копирование содержимого потока Srm1 в Srm2
        Srm2.CopyFrom(Srm1, Srm1.Size);
        writeln('File success copy');

    finally

```

```

//уничтожение объектов
Srm1.Free;
Srm2.Free;
end;

writeln('Press Enter to exit');
readln;
end.

```

### **Блок исключения**

Блок исключения имеет следующий синтаксис

```

try
  <операторы>
except
  <обработчики исключений>
else
  <операторы>
end;

```

Логика работы этого блока следующая. Если при выполнении операторов секции **try** **except** не было ошибок, то операторы в секции **except** **end** пропускаются. Если при выполнении секции **try** **except** происходит ошибка, то выполнении операторов этой секции прерывается, начиная с оператора вызвавшего ошибку и начинают выполняться операторы секции **except** **end** (рисунок 2).

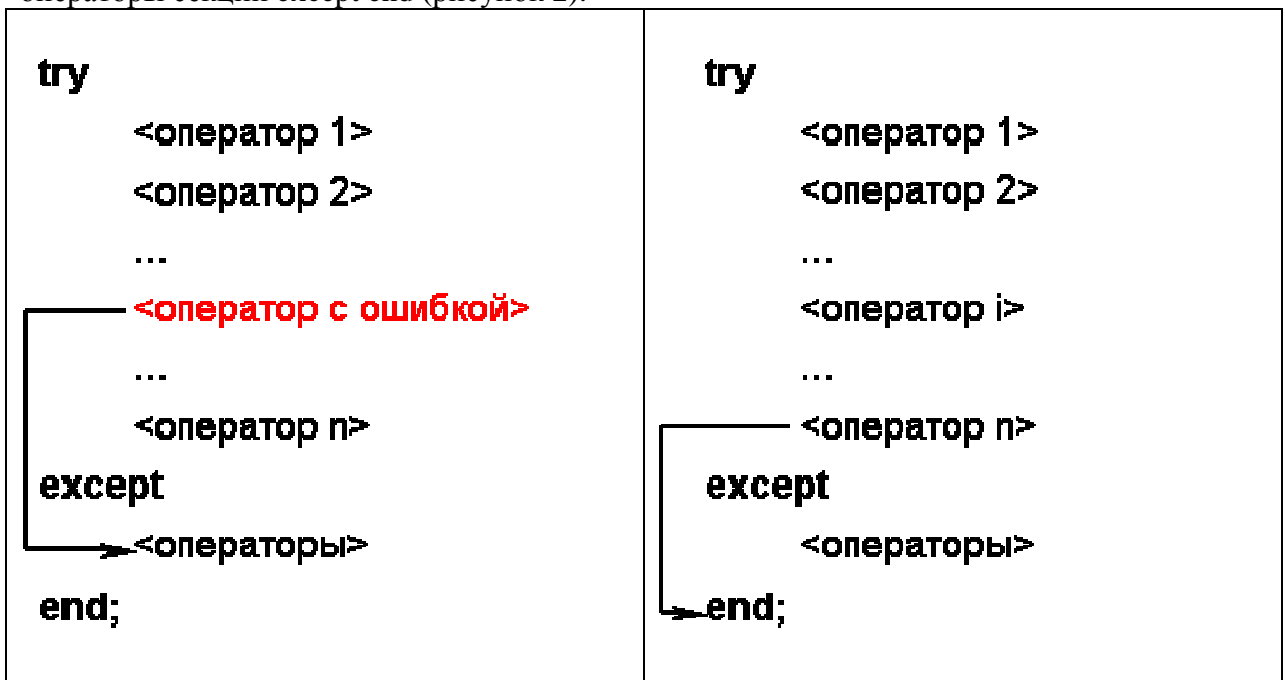


Рисунок 30 – Работа блока исключения

В секции **except** **end** располагаются обработчики исключений имеющие следующий синтаксис

```
on <класс исключения> do <оператор>;
```

где **on**, **do** – зарезервированные слова;

<класс исключения> – класс обработки исключения.

Имя класса служит ключом выбора, а обработка исключения производится оператором стоящим после слова `do`. Поиск нужного обработчика осуществляется с начала списка вниз до тех пор пока не встретится класс способный обрабатывать исключение данного типа. Если подходящего класса не встретится, то управление передается операторам, стоящим за `else`. Если тип и причина возникновения ошибки не существенны, то можно в секции `except end` вообще не указывать обработчики исключений, а сразу вставить код реакции на ошибку, например

```
try
...
except
...
writeln('Fatal Error')
...
end;
```

Защищенные блоки могут вкладываться друг в друга на неограниченную глубину.

Модифицируем программу приведенную в листинге 1

Листинг 69

```
program CopyFile2;
{$APPTYPE CONSOLE}
uses SysUtils, Classes;

var Srm1,                               //ВХОДНОЙ ПОТОК
    Srm2:TFileStream;                  //ВЫХОДНОЙ ПОТОК
    SourceFileName,                    //ИМЯ ВХОДНОГО ФАЙЛА
    DestFileName:string;              //ИМЯ ФАЙЛА КОПИИ
begin
    //открываем файл источник данных
    write('Source file >> ');
    readln(SourceFileName);

    try //в этом блоке может произойти ошибка
    try
        //создаем поток в режиме только для чтения
        Srm1:=TFileStream.Create(SourceFileName, fmOpenRead);

        //создаем файл приемник
        write('Destination file >> ');
        readln(DestFileName);
        //создаем поток в режиме создания нового потока
        Srm2:=TFileStream.Create(DestFileName, fmCreate);

        //копирование содержимого потока Srm1 в Srm2
        Srm2.CopyFrom(Srm1, Srm1.Size);
        writeln('File success copy');
    except
        writeln('Fatal Error!');
    end;
finally
    //уничтожение объектов
    Srm1.Free;
    Srm2.Free;
end;

writeln('Press Enter to exit');
readln;
end.
```

## Возбуждение исключений

Для инициирования собственного исключения необходимо использовать зарезервированное слово `raise` (возбудить). Если это слово встретилось в секции `try...exception` или `try...finally`, немедленно начинают свою работу секции соответственно `except...end` и `finally...end`. Если оно встретилось в `except...end` или `finally...end`, считается, что данный защищенный блок на текущем уровне вложенности (блоки могут быть вложенными) завершил свою работу и управление передается вышестоящему уровню.

Слово **raise** возбуждает исключение самого общего класса `Exception`. Если необходимо возбудить исключение конкретного типа (неважно - стандартного или собственного), необходимо явно указать класс создаваемого в этот момент объекта путем вызова его конструктора. Например, следующий оператор возбудит ошибку ввода/вывода:

```
raise EInOutError.Create('Ошибка!');
```

Обратите внимание, что после возбуждения исключительной ситуации и создания объекта для обработки исключительной ситуации, осуществляется переход на ближайший блок `except end`, т.е. операторы стоящие после **raise ...** не будут выполняться.

## Классы исключений

### Класс *Exception*

Класс `Exception` является родительским для всех классов исключений. Этот класс, а также многие его потомки описаны в модуле `SysUtils`. Класс `Exception` имеет несколько важных свойств и методов.

**constructor** `Create(const Msg: String);`

Создает объект исключения и помещает в него сообщение `Msg`.

**property** `Message: String`

Содержит сообщение об ошибке.

### Стандартные классы исключений

В Delphi определено много стандартных классов исключений, некоторые из них приведены в таблице 1.

Таблица 35 – Стандартные классы исключений

<i>Класс исключения</i>	<i>Родительский класс</i>	<i>Описание</i>
<code>EAbort</code>	<code>Exception</code>	Реализует «тихую» (без какого-либо сообщения) обработку любого исключения.
<code>EAbstractError</code>	<code>Exception</code>	Программа пытается вызвать абстрактный метод.
<code>EAccessViolation</code>	<code>Exception</code>	Программа пыталась обратиться к не принадлежащей ей области памяти или использует недействительный указатель.
<code>EControlC</code>	<code>Exception</code>	Возникает при нажатии <code>Ctrl-C</code> при работе приложения в режиме консоли.
<code>EConvertError</code>	<code>Exception</code>	Ошибка преобразования в функциях <code>StrToInt</code> или <code>StrToFloat</code> .



<i>Класс исключения</i>	<i>Родительский класс</i>	<i>Описание</i>
EDivByZero	EIntError	Ошибка целочисленного деления на ноль
EFCreateError	EStreamError	Ошибка при создании файла. Например, попытка создать файл на устройстве, предназначенном только для чтения, или в несуществующем каталоге.
EFOpenError	EStream-Error	Ошибка открытия потока данных. Например, попытка открыть несуществующий файл.
EInOutError	Exception	Любая ошибка в файловых операциях. Поле ErrorCode объекта этого класса содержит код ошибки.
EIntError	Exception	Любая ошибка в целочисленных вычислениях.
EInvalidCast	Exception	Программа пытается осуществить недопустимое преобразование типов с помощью оператора as.
EInvalidOp	EMatchError	Ошибка в операциях с плавающей точкой (недопустимая инструкция, переполнение стека сопроцессора и т.п.).
EMatchError	Exception	Любая ошибка при выполнении вычислений с плавающей точкой.
EOutOfMemory	Exception	Эта ошибка возникает, когда программа запрашивает слишком большой для данной конфигурации Windows объем памяти.
EOverflow	EMatchError	Результат операций с плавающей точкой слишком велик, чтобы уместиться в регистрах сопроцессора.
EWriteError	EFileError	Ошибка записи в поток данных.
EZeroDivide	EMatchError	Вещественное деление на ноль.

Обратите внимание, что ищется **самый первый** из, возможно, нескольких обработчиков, **класс** которого **способен обрабатывать данное исключение**. Если, например, в списке первым стоит EAbort, который может обработать любое исключение, ни один из стоящих за ним обработчиков никогда не получит управления. Точно так же, если указан обработчик для класса EIntError, за ним бесполезно размещать обработчики EDivByZero, ERangeError или EIntOverflow:

При возникновении исключительной ситуации объекты классов-обработчиков создаются и уничтожаются автоматически, т.е. являются объектами с управляемым временем жизни. Если необходимо использовать поля или методы класса-обработчика явно, необходимо поименовать автоматически создаваемый объект. Для этого перед именем класса ставится идентификатор и двоеточие:

```
try
...
except
...
on E: EClassName do ; writeln(E.Message);
```


```
end;
```

Для стандартных классов такой прием фактически позволяет использовать единственное строковое свойство Message со стандартным сообщением об ошибке, которое получают все наследники класса Exception.

## Создание собственного класса исключений

Можно создать собственный класс обработки исключений, объявив его потомком Exception или любого другого стандартного класса. Объявление нестандартного класса имеет смысл только тогда, когда необходимо научить программу распознавать некорректные наборы данных и соответствующим образом на них реагировать. Например

```
type EMyError=class(Exception);
```

 **Совет:** В Delphi принято для имен классов исключений использовать префикс «E». Рекомендуется при создании собственных классов исключений руководствоваться этим же принципом. Примеры имен классов исключений можно посмотреть в таблице 1.

Ниже приведен пример программы использующей пользовательский обработчик исключительной ситуации. Программа обрабатывает две исключительные ситуации неверный формат числа (EInOutError) и выражение под квадратным корнем меньше нуля (EMyError).

Листинг 70

```
program Myexp;

{$APPTYPE CONSOLE}

uses SysUtils;

// класс исключения
type EMyError=class(Exception);

function Calculate(x:real):real;
begin
  if 1-x*x<0 then raise EMyError.Create('Demo MyError');
  result:=sqrt(1-x*x);
end;

var value:real;
begin
  try
    write('value > ');
    readln(value);
    writeln('result=', calculate(value));
  except
    on Err:EMyError do writeln('Error: ', Err.message);
    on Err:EInOutError do writeln('Error: ', Err.message);
  else
    writeln('Unknown Error');
  end;

  writeln('Press Enter to Exit');
  readln;
end.
```

## Задания к лабораторной работе

1. Модифицируйте программы из заданий 2-5 лабораторной работы № 6, добавив проверку на ошибки ввода-вывода с помощью защищенных блоков
2. Модифицируйте программу из листинга 12 лабораторной работы № 11, добавив проверку на ошибки ввода-вывода с помощью защищенных блоков
3. Модифицируйте функцию SearchNode из модуля Lists (лабораторная работа № 10) таким образом, чтобы при отрицательном результате поиска ( т.е. заданный элемент в списке не найден) возникала исключительная ситуация ENotFoundError. Модифицируйте также программу из листинга 1 (лабораторная работа № 10) так, чтобы она могла обрабатывать исключительную ситуацию ENotFoundError, а также исключительные ситуации связанные с ошибками ввода-вывода.

## Вопросы к лабораторной работе

1. В чем суть механизма обработки исключительных ситуаций?
2. Что такое защищенный блок?
3. Как работает защищенный блок завершения?
4. Как работает защищенный блок исключения?
5. Как программно возбудить исключение?
6. Какие стандартные классы исключений вы знаете?
7. Как создать свой класс исключение?
8. Какой особенностью работы с памятью обладают классы исключений?
9. Как получить доступ к полям и методам класса исключения во время обработки ошибки в секции except end?
10. Какой алгоритм используется при выборе исключения в секции except end?

## Справочные таблицы

Таблица 1– Стандартные классы исключений .....177

# Лабораторная работа №13

## Отладка и трассировка программ.

### Ведение протокола работы программы.


#### Введение

В лабораторной работе рассмотрены основные возможности отладчика **Delphi**. Рассмотрены примеры применения точек останова, окна наблюдения, окна **Evaluate/Modify**.

#### Отладка программ





В Delphi имеется мощный встроенный отладчик, значительно упрощающий отладку программ. Основными инструментами отладки являются точки контрольного останова и окно наблюдения за переменными.

#### Прерывание работы программы

Если программа запущена из среды Delphi, ее работу можно прервать в любой момент с помощью клавиш <Ctrl>+<F2>, кнопки  или команды **Run ► Program pause**.

#### Трассировка программы

Трассировкой программы называется пошаговое выполнение программы. При трассировке остановка программы происходит после выполнения каждого оператора (строки). При этом управление получает среда Delphi, а в окне наблюдения отражается текущее значение наблюдаемых переменных и/или выражений.

Трассировку программы можно выполнять с помощью клавиш <F7> и <F8> или кнопок  и  панели инструментов соответственно. При нажатии клавиши <F8> будут выполнены запрограммированные в текущей строке действия, и работа программы прервется перед выполнением следующей строки текста. При нажатии клавиши <F7> среда выполняет те же действия, что и при нажатии клавиши <F8>, однако если в текущей строке содержится вызов подпрограммы пользователя, программа прервет свою работу перед выполнением первой исполняемого оператора этой подпрограммы, т.е. клавиша <F7> позволяет трассировать вызываемые подпрограммы. Текущая выполняемая строка операторов выделяется синим цветом кроме того, признаком текущей строки является особый символ в служебной зоне слева в окне редактора . После трассировки нужного фрагмента программы можно продолжить нормальную ее работу, нажав клавишу <F9> или кнопку .

#### Точки останова

Точка контрольного останова определяет оператор в программе, перед выполнением которого программа прервет свою работу, и управление будет передано среде **Delphi**. Точка останова задается командой **View ► Debug windows ► Breakpoints**. Контрольная точка останова выделяется по умолчанию красным цветом (Рисунок 1).

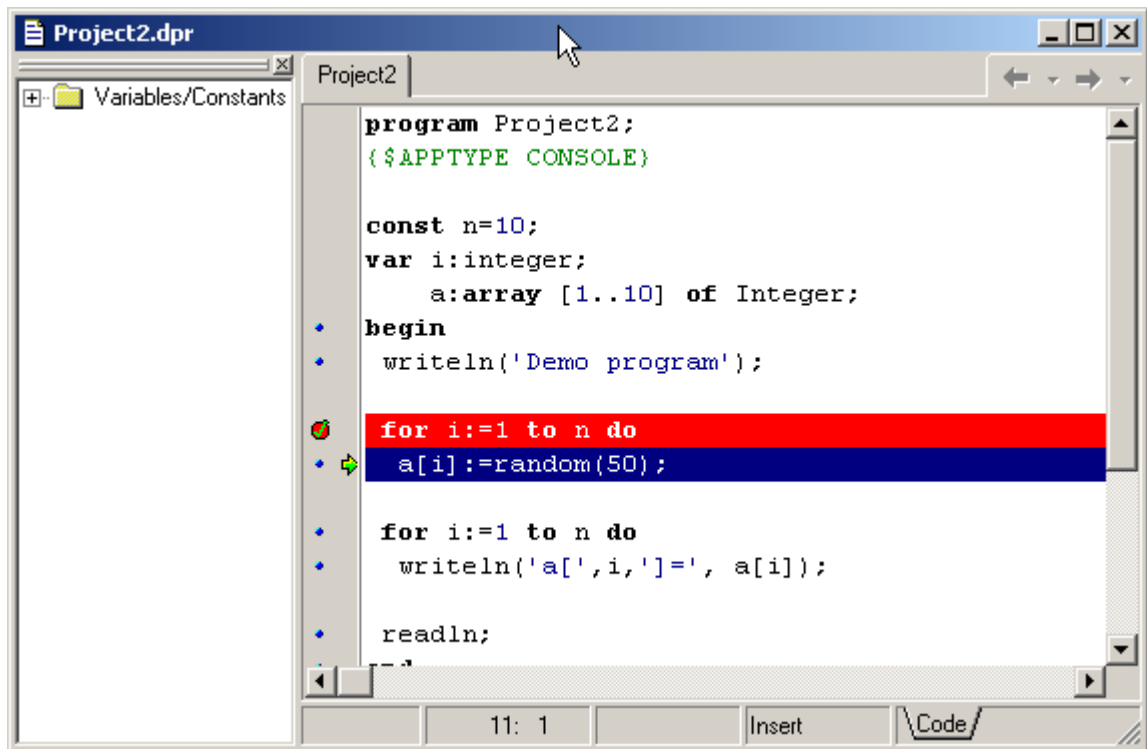


Рисунок 31 – Вид окна редактора кода при срабатывании точки останова.

Окно точек останова (рисунок 2) содержит список всех установленных в проекте точек, перед выполнением которых происходит прекращение работы программы и управление получает среда **Delphi**.

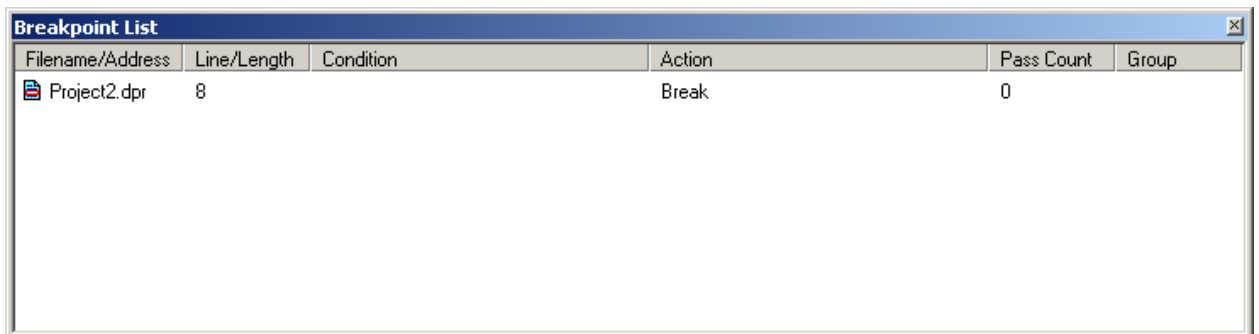


Рисунок 32 – Окно списка точек останова

Чтобы добавить новую точку останова необходимо выбрать в контекстном меню команду **Add**. В этом случае появляется, окно (рисунок 3), с помощью которого можно задать параметры добавляемой точки.

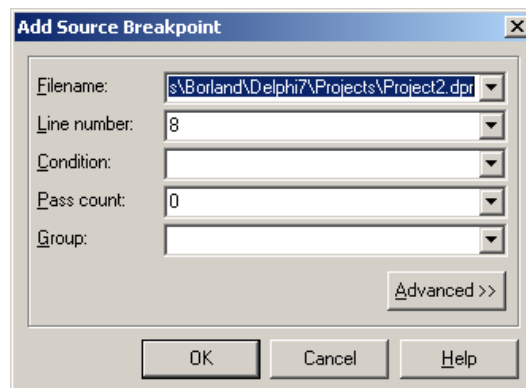




Рисунок 33 - Окно добавления точки останова

Окно приведенное на рисунке 3 содержит следующие поля:

- **Filename** – путь и имя файла (в момент появления окна это поле содержит имя открытого файла);
- **Line number** – номер строки от начала файла (в момент появления окна это поле содержит номер строки в которой находится курсор);
- **Condition** – в этом поле можно указать условие останова в виде, логического выражения;
- **Pass count** – в этом поле можно указать количество проходов, программы через контрольную точку без прерывания вычислений.

 **Совет:** Для того чтобы установить/снять точку контрольного останова, достаточно щелкнуть мышью в служебной зоне слева от нужной строки или установить в эту строку текстовый курсор и нажать клавишу <F5>.

В Delphi версии 5, 6 и 7 с любой точкой можно связать одно или несколько действий. Для этого нужно активизировать окно точек останова, вызвать его контекстное меню (щелчок правой кнопкой мыши на точке останова) и выбрать команду **Properties**. В появившемся окне свойств точек останова необходимо нажать кнопку **Advanced** , чтобы получить доступ к дополнительным свойствам (рисунок 4).

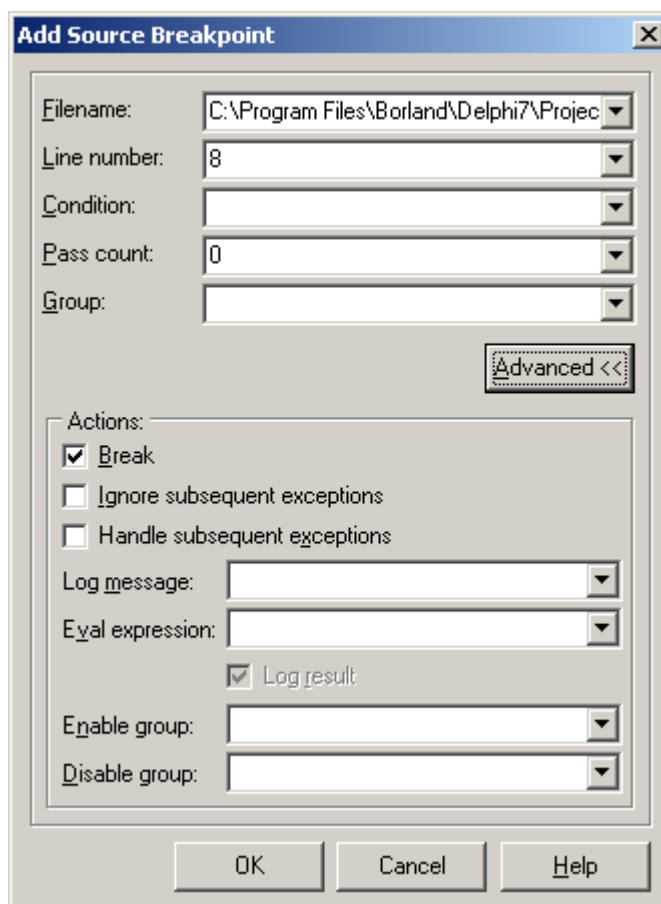


Рисунок 34 – Дополнительные параметры точки останова

В нижней части окна появится группа Actions, с помощью которой определяются действия для точки останова, указанной в верхней части окна:

- **Break** – простой останов перед выполнением помеченного оператора;

- **Ignore subsequent exceptions** – если флажок установлен, игнорируются все возможные последующие исключения в текущем отладочном сеансе до очередной точки останова, в которой, возможно, это действие будет отменено;
- **Handle subsequent exceptions** – после установки этого флажка отменяется действие предыдущего флажка и возобновляется обработка возможных исключений;
- **Log message** – этот список позволяет выбрать произвольное сообщение, связанное с точкой останова;
- **Eval expression** – Позволяет вычислить некоторое выражение и поместить его результат в сообщение, выбранное в списке **Log message**.

В качестве примера рассмотрим отладку программы из листинга 1.

Листинг 71

```

program Project2;
{$APPTYPE CONSOLE}

const n=10;
var i:integer;
    a:array [1..10] of Integer;
begin
  writeln('Demo program');

  for i:=1 to n do
    a[i]:=random(50);

  for i:=1 to n do
    writeln('a[' ,i, ']=' , a[i]);

  readln;
end.

```

Установим точку останова как показано на рисунке 5.

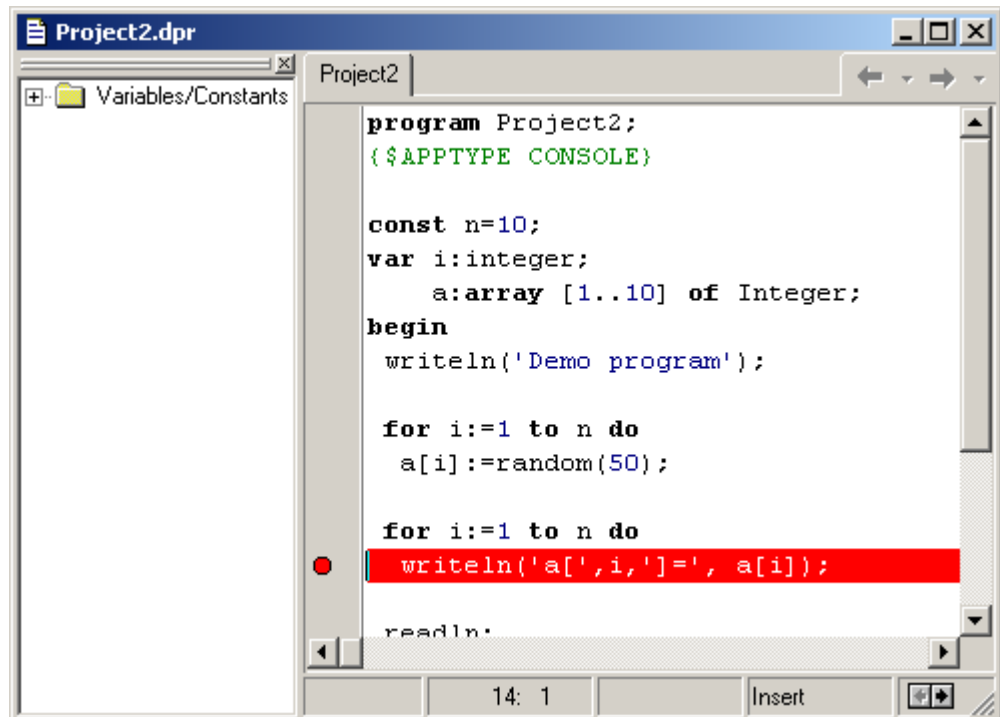


Рисунок 35

Изменим, параметры этой точки останова так, чтобы она срабатывала только при значениях счетчика цикла больше 5. Для этого необходимо установить свойство **Pass Count** как на рисунке 6.

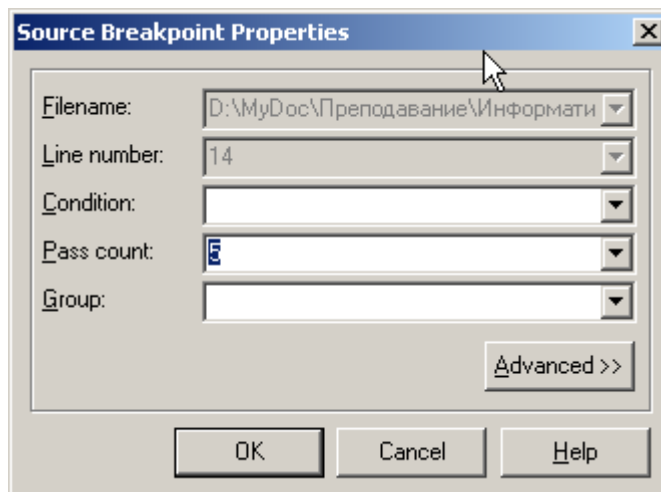


Рисунок 36

После запуска программы на выполнение отмеченная строка будет выполнена пять раз затем программа будет остановлена и управление будет передано отладчику.

## Группировка точек останова

В Delphi версии 5, 6 и 7 существует возможность объединения точек останова в группы. Для этого используется окно настройки параметров точки останова (рисунок 3). В раскрывающемся списке **Group** следует выбрать имя группы, к которой принадлежит точка, а с помощью списков **Enable group** и **Disable group** соответственно разрешить или запретить действие всех точек останова, относящихся к выбранной группе.

## Ведение протокола работы программы

В ряде случаев бывает неудобно или невозможно пользоваться пошаговой отладкой программ. В таких ситуациях можно применять контрольные точки, которые не прерывают работу программы, а лишь помещают некоторую информацию в специальный файл трассировки. Для реализации такой точки необходимо открыть окно **Source Breakpoint Properties** (рисунок 4), снять флажок **Break** и ввести сообщение в поле раскрывающегося списка **Log message**. Можно также в поле списка **Eval expression** ввести некоторое выражение, которое будет вычислено и вместе с сообщением помещено в протокол работы программы. Этот протокол можно просмотреть в любой момент (в том числе и после завершения прогона программы) с помощью команды **View ► Debug Windows ► Event Log** (рисунок 7).



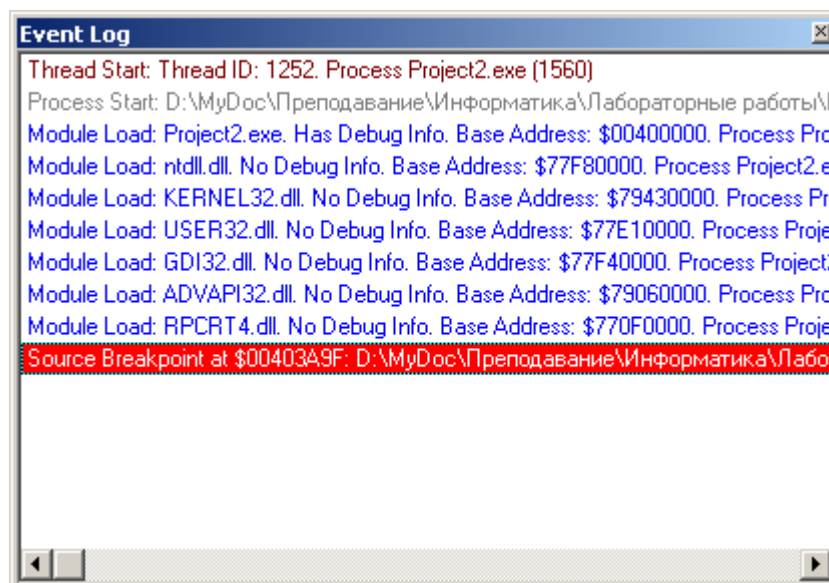


Рисунок 37 – Окно протокола работы программы

Рассмотрим пример ведения протокола программы на предыдущем примере. Введите данные в окно свойств точки останова как на рисунке 8.

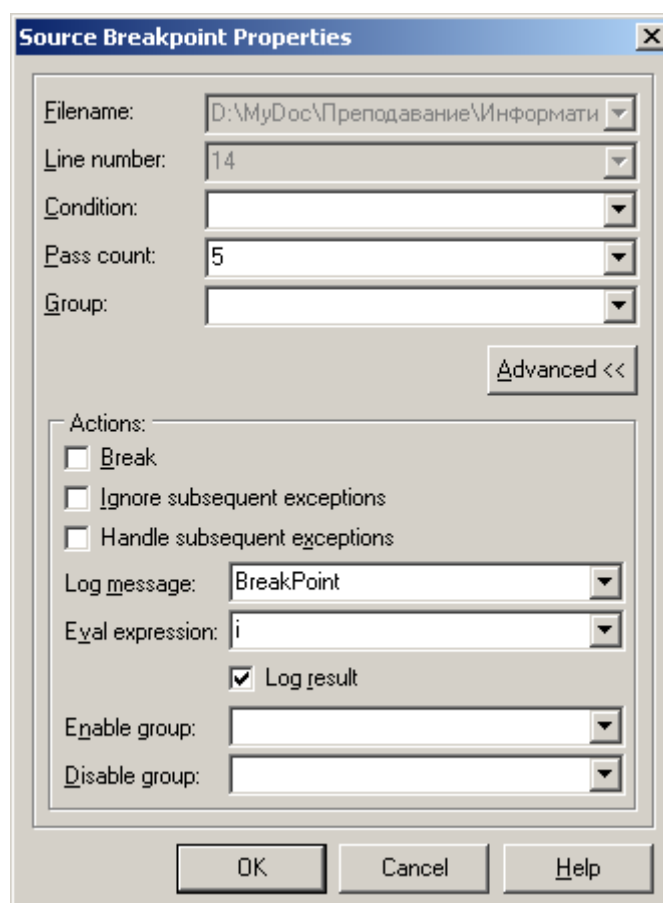


Рисунок 38

Откомпилируйте программу и откройте окно протокола программы, его вид будет примерно таким как на рисунке 9.

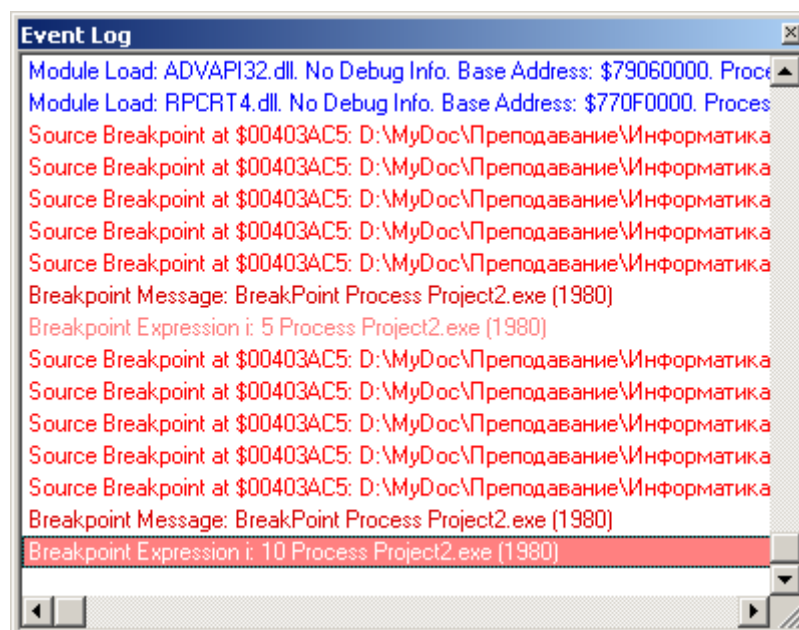


Рисунок 39

## Окно наблюдения (Watch List)

Наблюдать за состоянием переменной или выражения можно с помощью специального окна (рисунок 10), вызываемого командой **View ► Debug windows ► Watches**.

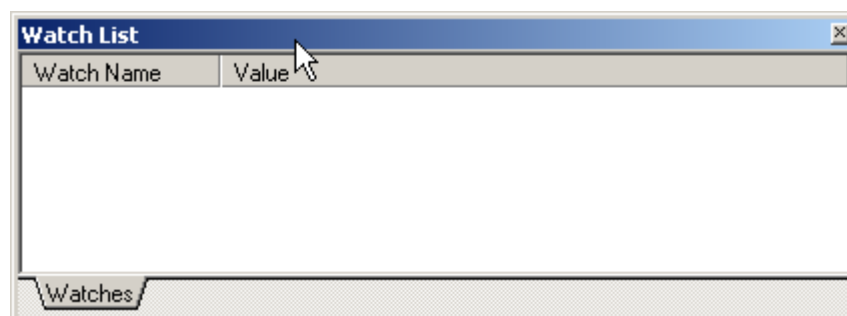


Рисунок 40 – Окно наблюдения

Окно наблюдения используется в отладочном режиме для наблюдения за изменением значений выражений, помещенных в это окно. Для добавления нового выражения необходимо вызвать контекстное меню окна наблюдения и выбрать команду **New Watch**. В поле раскрывающегося списка **Expression** (рисунок 11) вводится выражение. Поле **Repeat count** определяет количество показываемых элементов массивов данных; поле **Digits** – количество значащих цифр для отображения вещественных данных; флажок **Enabled** разрешает или запрещает вычисление, выражения. Остальные элементы определяют способ представления значения.

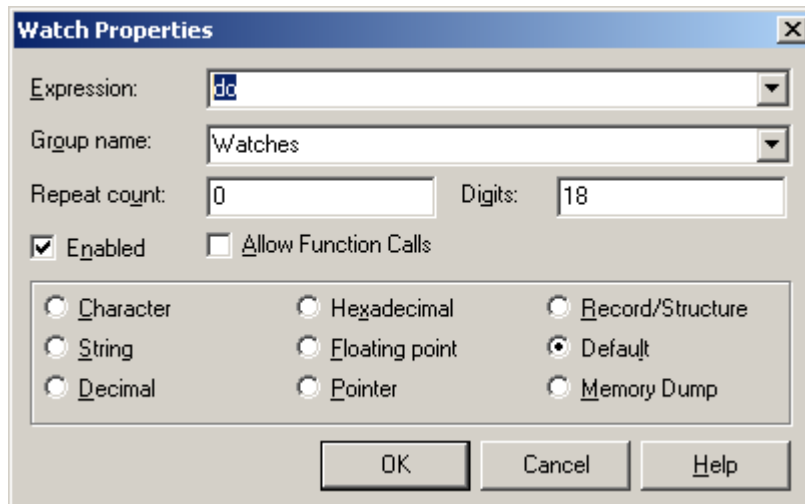



Рисунок 41 – Окно настройки окна наблюдения

 **Совет:** Для просмотра текущих значений любых переменных в отладочном режиме, достаточно задержать на переменной указатель мыши – значение появится во всплывающей подсказке рядом с указателем (рисунок 12).

```

• begin
•   writeln('Demo program');
•   for i:=1 to n do
•     a[i]:=random(50);
•     a = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
•   for i:=1 to n do
•     writeln('a[' , i, ']= ' , a[i]);

```

Рисунок 42 – Просмотр текущего состояния переменной

## Вычисление и изменение значений выражений

С помощью окна **Evaluate/Modify** (рисунок 13) можно изменить значение любой переменной. Это окно можно вызвать в режиме отладки с помощью клавиш Ctrl+F7.

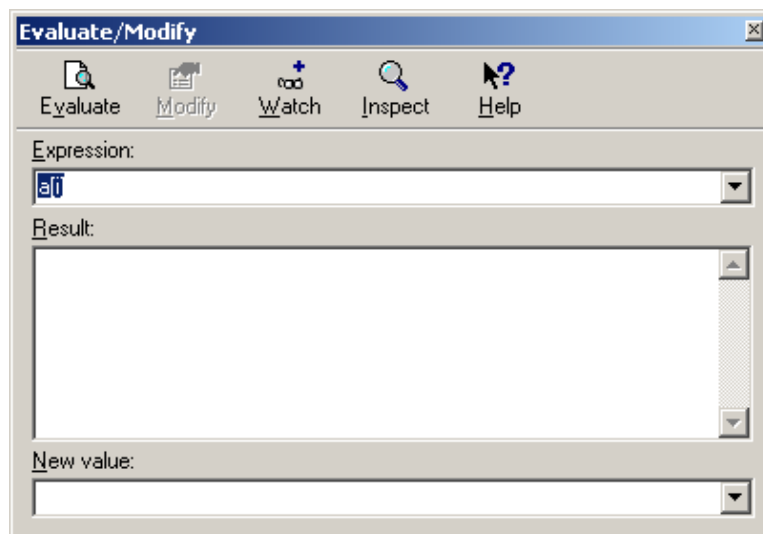





Рисунок 43

В поле списка **Expression** вводится имя переменной значение которой нужно изменить или просмотреть. После нажатия кнопки **Evaluate**  в поле **Result** появится текущее значение переменной. Для изменения значения переменной введите новое значение переменной в поле **New value** и нажмите кнопку **Modify** . Кнопка  **Watch** служит для переноса переменной в окно наблюдения **Watch**.

## Окно Local Variables

Для отслеживания и изменения значений локальных переменных удобнее применять окно **Local Variables** (рисунок 14). В окне отображаются текущие значения всех локальных переменных подпрограммы.

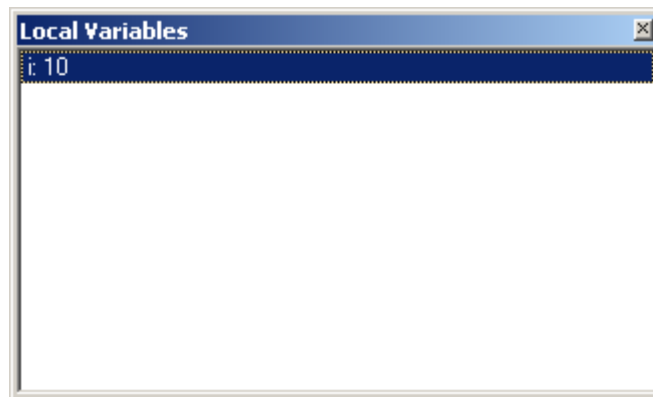


Рисунок 44 – Окно для просмотра значений локальных переменных

Для изменения значения переменной дважды щелкните на окне **Local Variables**, после этого появится окно **Debug Inspector** (рисунок 15).

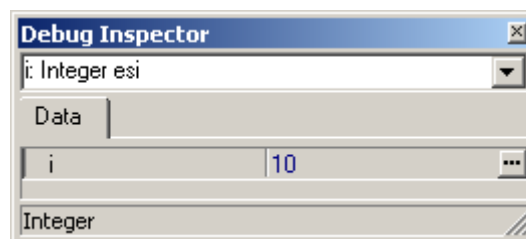



Рисунок 45 – Окно Debug Inspector

Для изменения значения переменной необходимо нажать кнопку  напротив идентификатора переменной. После этого откроется окно **Change** изменения значения переменной (рисунок 16).

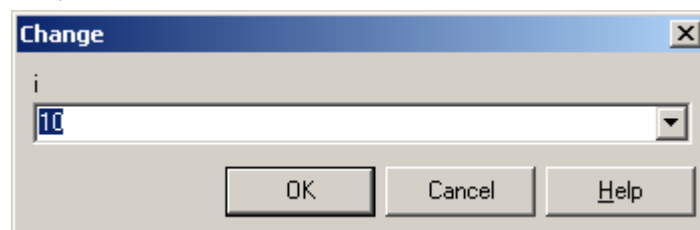


Рисунок 46 – Окно Change

Необходимо ввести новое значение переменной и нажать кнопку **Ok**.

## Вопросы к лабораторной работе

1. Что такое трассировка программы, как она выполняется?

2. Что такое точка останова, как задать точку останова?
3. Как задать или изменить параметры точки останова?
4. Как организовать ведение протокола программы с помощью точек останова?
5. Какие действия можно выполнять с помощью окна наблюдения?
6. Как изменить значение переменной во время отладки программы?
7. Каково назначение окна Local Variables?

Учебное издание

*Еленев Валерий Дмитриевич,*

*Гоголев Михаил Юрьевич*

**АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ И ТЕХНОЛОГИИ  
ПРОГРАММИРОВАНИЯ НА ЯЗЫКАХ  
ВЫСОКОГО УРОВНЯ**

*Лабораторный практикум*

В авторской редакции

Подписано в печать 03. 06. 2010. Формат 60x84 1/16.

Бумага офсетная. Печать офсетная. Печ. л. 12,0.

Тираж 50 экз. Заказ .

Самарский государственный  
аэрокосмический университет.  
443086, Самара, Московское шоссе, 34.

---

Изд-во Самарского государственного  
аэрокосмического университета. 443086,  
Самара, Московское шоссе, 34.



