

Проф. И.Г. Чугреев

Учебное пособие по программированию в среде Pascal ABC NET

Учебное пособие для студентов геодезических специальностей



МИИГАиК

2023 г.

Введение

В 2003 году на факультете математики, механики и компьютерных наук Южного Федерального Университета была создана учебная среда программирования Pascal ABC.

PascalABC.NET — среда программирования нового поколения, включающая классический Паскаль и большинство возможностей языка Delphi. В данной среде можно программировать в структурном, объектно-ориентированном и функциональном стилях.

PascalABC.NET позиционируется разработчиками как среда программирования для сферы образования и научных исследований.

9 марта 2022 г. вышла версия PascalABC.NET 3.8.3.

В данном пособии будет рассмотрено объектно-ориентированное программирование задач под WINDOWS.

Знакомство со средой программирования.

PascalABC.NET можно загрузить с официального сайта разработчика бесплатно. Лицензия не требуется. После установки системы на компьютер, на рабочем столе появится пиктограмма –



Для загрузки среды разработки программного обеспечения необходимо дважды щелкнуть мышкой по данной пиктограмме. В результате получим окно для разработки консольных программ Рис.1:

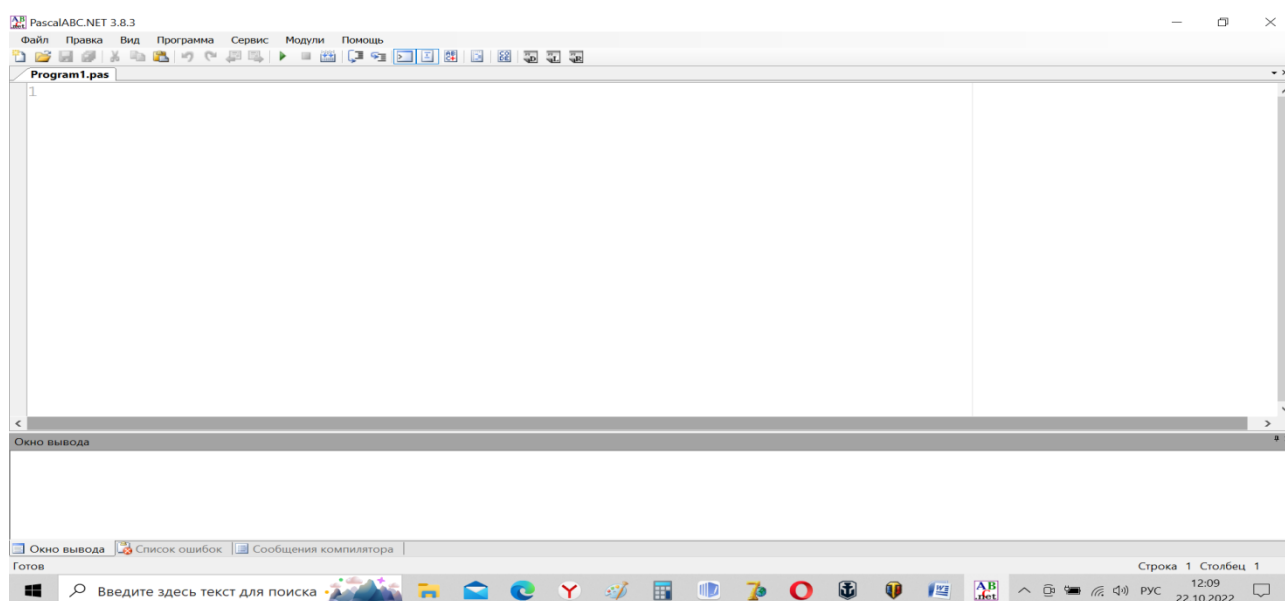


Рис.1

Данное окно состоит из

1. строчного меню и «горячих кнопок» рис.2,

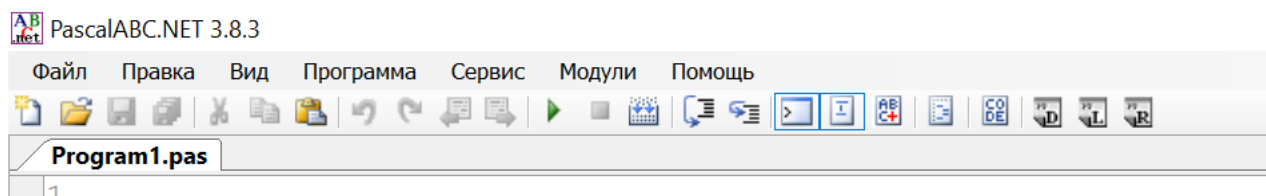


Рис.2

которые дублируют свойства строчного меню. Например, кнопка строчного меню «Файл», представленная на рис.3 в раскрытом виде – ее дублируют первые 4 «горячих» кнопки, наиболее часто используемые

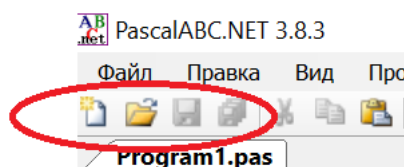
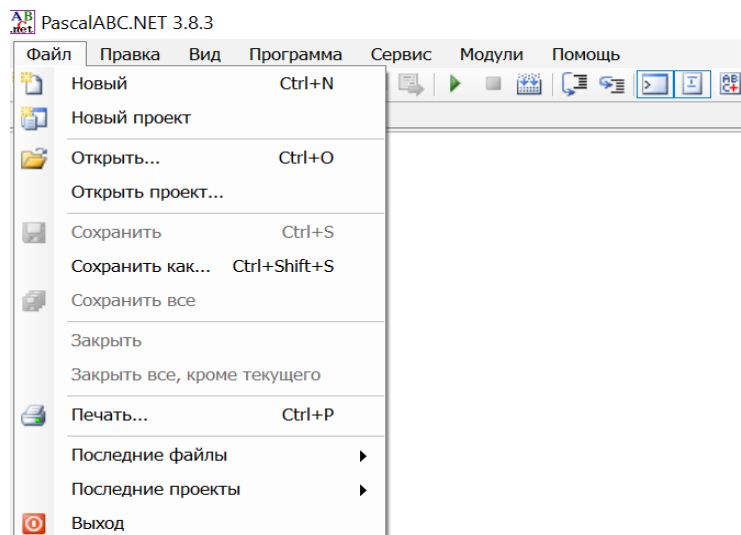
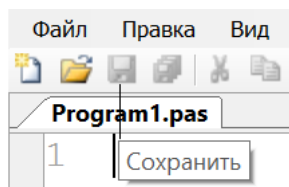


Рис.3

при разработке приложений – **новый, открыть** (файл на диске), **сохранить, сохранить все**. При наведении курсора на «горячую» кнопку появляется ярлычок с подсказкой о назначении данной кнопки



Следующие 5 кнопок дублируют меню «Правка» – вырезать, копировать, вставить, отменить, повторить...

2. Окна разработки кода (в консольном режиме) рис.4

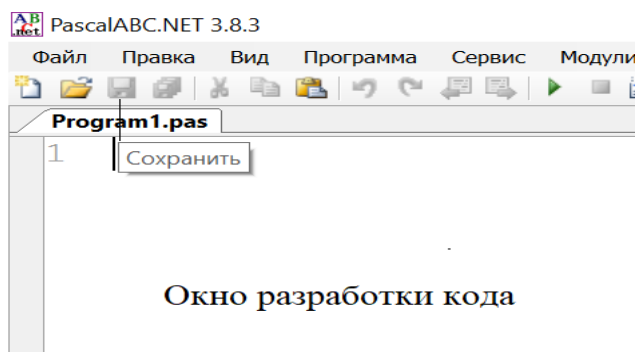


Рис.4

3. Окна списка ошибок, при компиляции программы. Рис.5

<		
Список ошибок		
	Строка	Описание

Рис.5

Создания приложений под Windows в среде PascalABC.NET.

Для разработки нового приложения под Windows необходимо выполнить следующие действия:

1. Войти в меню «Файлы» и выбрать кнопку «Новый проект»

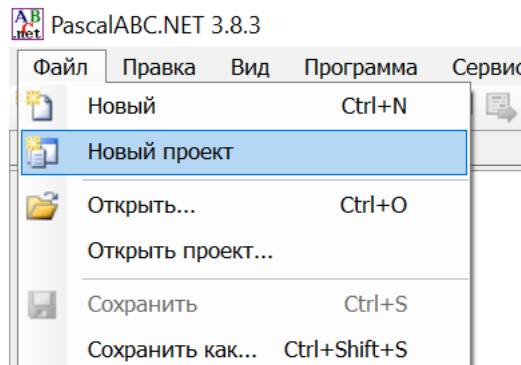


Рис. 6

(Для открытия существующего проекта выбрать кнопку «Открыть проект».)

2. При нажатии на кнопку «Новый проект» – рис.6 – откроется окно «Новый проект».

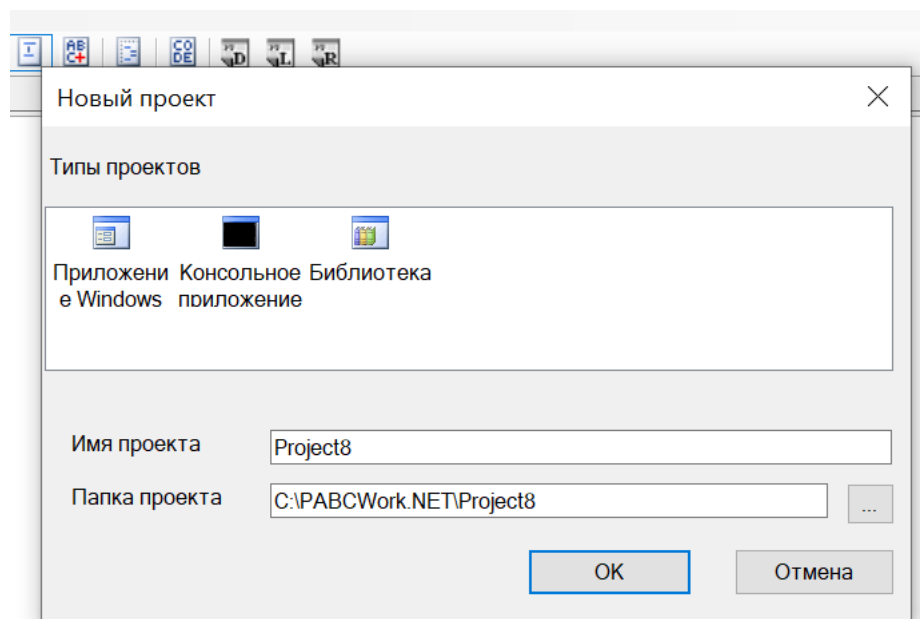


Рис.7

В данном окне (рис.7) необходимо задать путь к папке, в которой будет храниться проект,

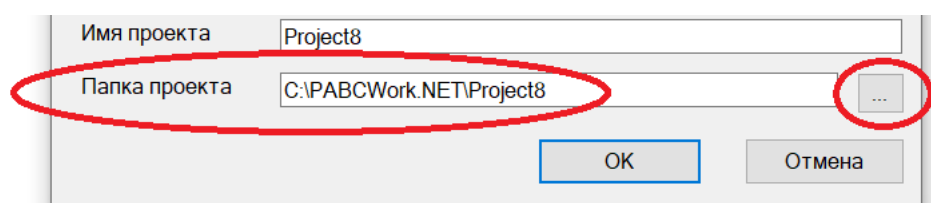


Рис.8

нажав на бутончик справа - рис. 8. При этом откроется модальное окно для работы с файлами и папками – рис.9

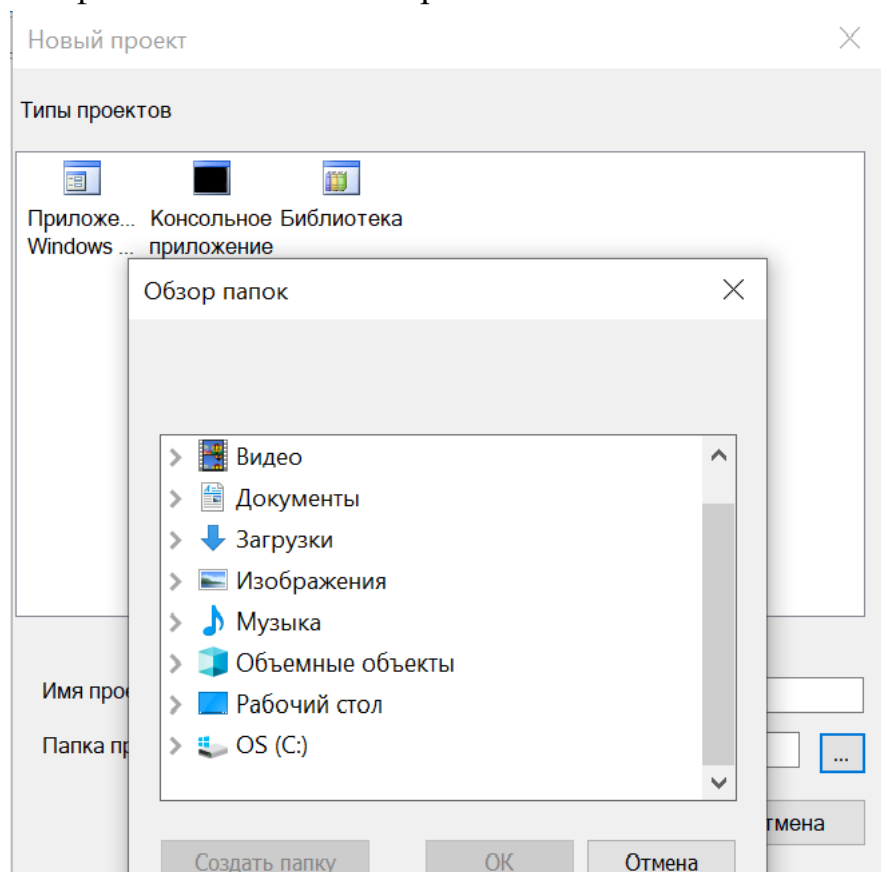


Рис.9

В данном окне необходимо выбрать диск, на котором создадим папку под проект. Все Ваши разработки по данному проекту будут храниться в данной папке. При необходимости, можно задать имя проекта – Рис.10

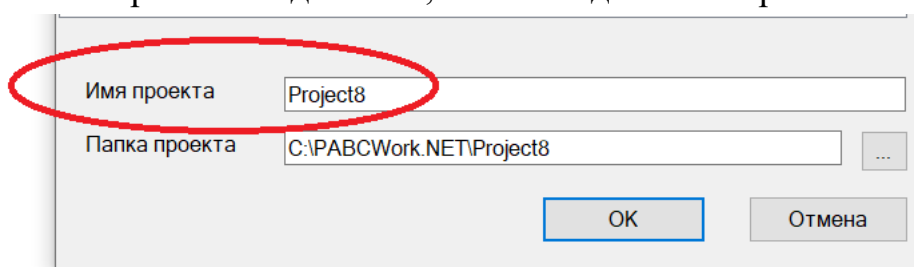


Рис.10

После чего выбираем пиктограмму «Приложение в Windows» - Рис.11

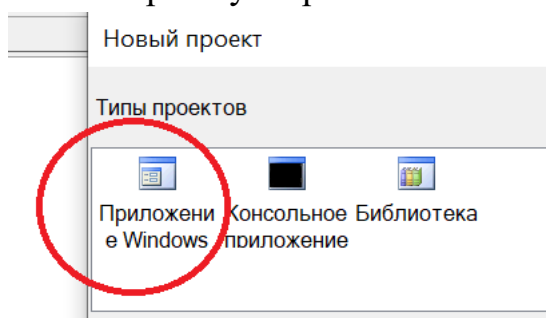


Рис.11

И нажимаем кнопку «Ok».

В результате загрузится окно для разработки приложений под Windows – Рис. 12 – в режиме «Дизайнер».

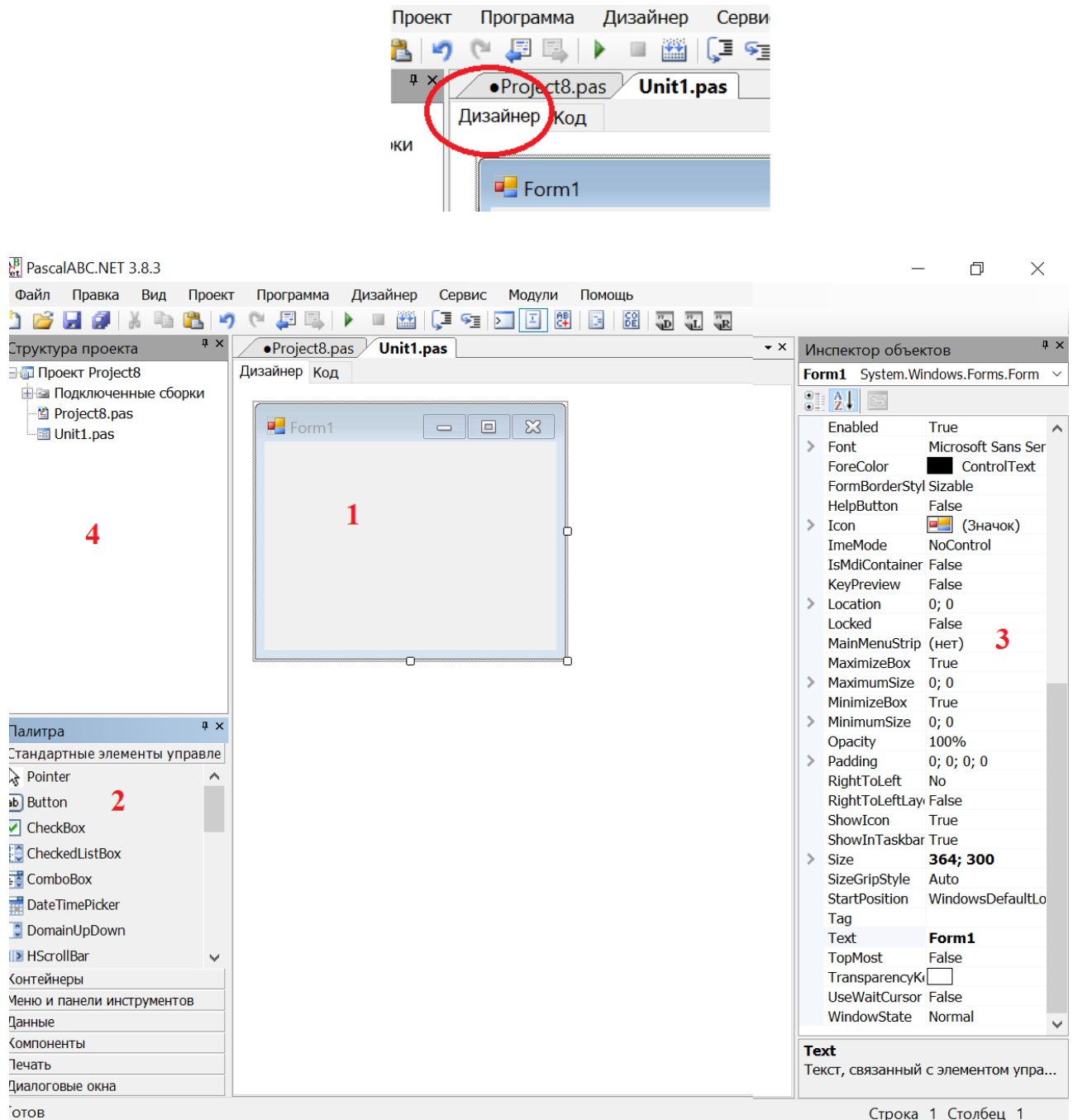


Рис.12

В данном окне (пронумеровано цифрами) присутствуют:

1. Form1 – рабочий стол, на котором разрабатывается интерфейс программы.
2. Палитра – содержит визуальные компоненты для разработки интерфейса программы и состоит из набора:
 - а) Стандартных элементов управления;
 - б) Меню и панели инструментов;
 - в) Данные;
 - г) Компоненты;
 - д) Печать;

- е) Диалоговые окна;
- ж) Контейнеры.

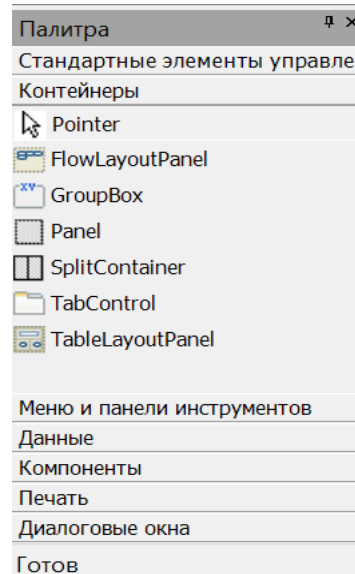


Рис.13

При нажатии на кнопку определенного набора отображаются соответствующие компоненты. На рис.13 представлен набор контейнеров.

Если при выборе компонента вы сделали ошибку, то нажатие курсором на стрелку «Pointer» отменит ошибочный выбор.

3. Инспектор объектов – рис.14

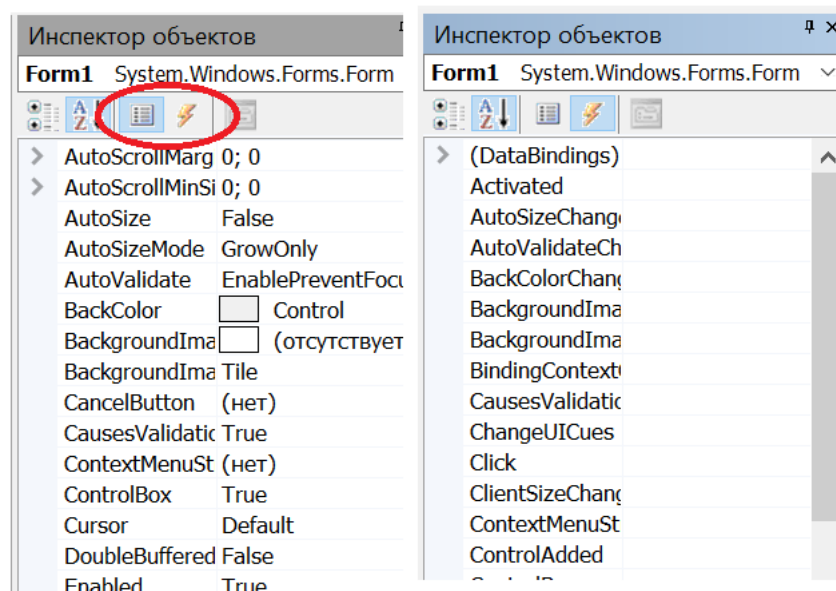


Рис.14

Инспектор объектов состоит из двух окон, одно из которых скрыто:

- **свойства** визуальных компонентов – левое окно
- и окно **события**, которые могут происходить с компонентами – правое окно.

В данном случае свойства и события применимы к компоненту **Form1**, о чем свидетельствует подсказка во второй строке таблицы.

Переключение между окнами осуществляется двумя кнопками, обведенные красным эллипсом.

Например:

Свойство - AutoSize – автоматическое задание размера компонента.

Событие - Click – создает обработчик события щелчка мышки по компоненту.

(Прежде чем создать обработчик события, рекомендуется сохранить проект, чтобы не потерять отредактированную часть данных)

4. Структура проекта.

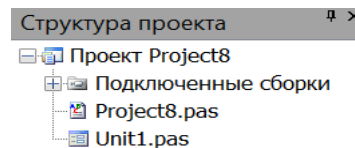


Рис.15

Содержит сборку проекта и все вложенные Unit_ы, входящие в проект.

5. Для перехода на страницу разработки кода необходимо нажать кнопку «Код»

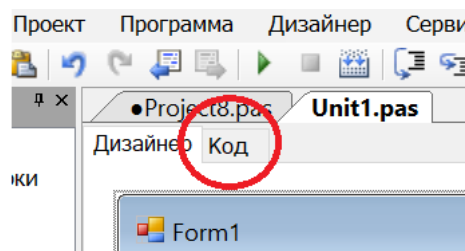


Рис.16

в результате чего на экране появится листинг (Unit) разработки кода –
Рис.17:

Unit содержит 2 раздела :

Interface

и

implementation

Interface – в данном разделе автоматически, без вмешательства программиста, формируется описательная часть программы (Unit_a) .

Uses - после данного объявления описываются подключаемые динамические библиотеки и все модули (Unit_ы) входящие в проект.

Type - в данном разделе формируется описание обработчиков событий процедурного типа для данного Unit_a.

Internal – в данном разделе описываются все компоненты, находящиеся на форме.

public - в данном разделе происходит инициализация всех компонентов проекта при его запуске.

Implementation - раздел разработки кода программы.

End. - с точкой, закрывает раздел Implementation.

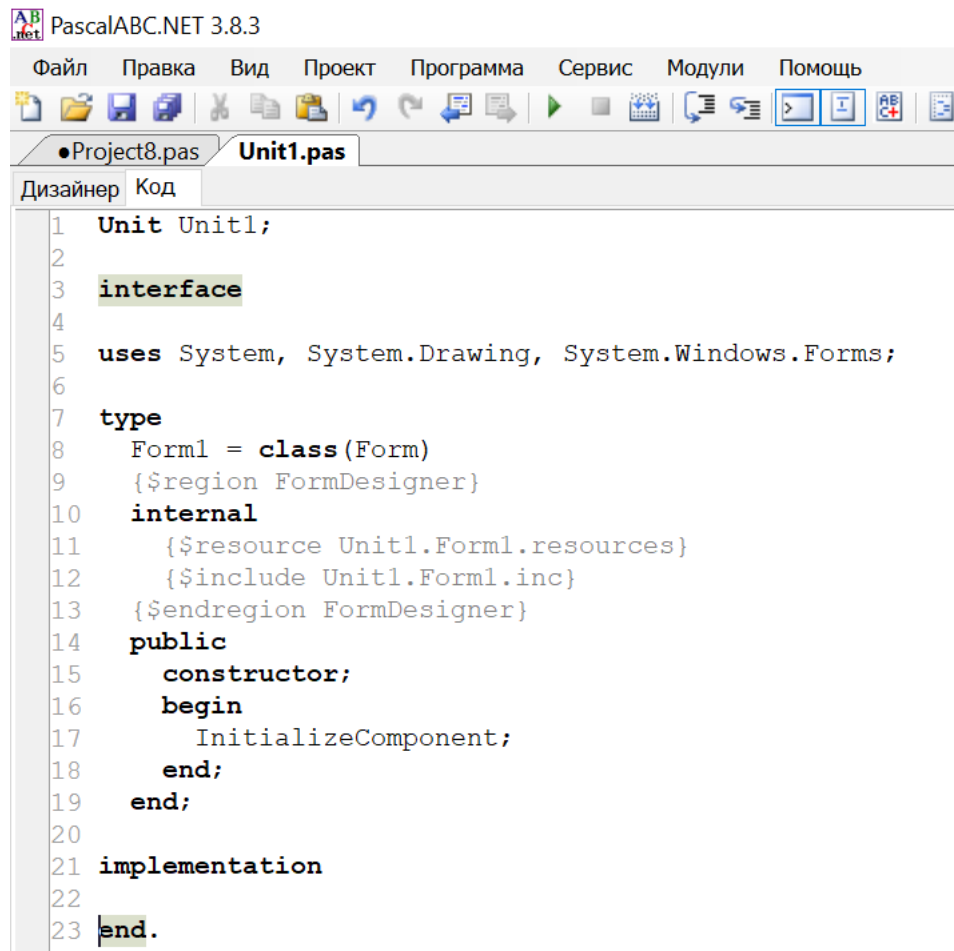


Рис.17

Все готово к программированию Ваших задач.

Далее познакомимся с основами программирования на языке Паскаль.

Основы программирования на языке Паскаль.

Основные символы языка

Основные символы языка - буквы, цифры и специальные символы составляют алфавит языка.

ПАСКАЛЬ включает следующий набор основных символов:

1. 26 латинских строчных и 26 латинских прописных букв:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

2. _ подчеркивание

3. 10 цифр:

0 1 2 3 4 5 6 7 8 9

4. знаки операций:

+ - * / = <> < > <= >= := @

5. ограничители:

. , ' () [] (.) { } (* *) .. : ;

6. спецификаторы:

^ # \$

7. служебные (зарезервированные) слова:

ABSOLUTE	EXPORTS	LIBRARY	SET
ASSEMBLER	EXTERNAL	MOD	SHL
AND	FAR	NAME	SHR
ARRAY	FILE	NIL	STRING
ASM	FOR	NEAR	THEN
ASSEMBLER	FORWARD	NOT	TO
BEGIN	FUNCTION	OBJECT	TYPE
CASE	GOTO	OF	UNIT
CONST	IF	OR	UNTIL
CONSTRUCTOR	IMPLEMENTATION	PACKED	USES
DESTRUCTOR	IN	PRIVATE	VAR
DIV	INDEX	PROCEDURE	VIRTUAL
DO	INHERITED	PROGRAM	WHILE
DOWNT0	INLINE	PUBLIC	WITH
ELSE	INTERFACE	RECORD	XOR
END	INTERRUPT	REPEATEXPORT	
LABEL	RESIDENT		

Кроме перечисленных, в набор основных символов входит пробел. *Пробелы нельзя использовать внутри сдвоенных символов и зарезервированных слов.*

Концепция типов данных

В любом алгоритмическом языке каждая константа, переменная, выражение или функция бывают определенного типа.

В языке ПАСКАЛЬ существуют скалярные и структурированные типы данных.

К скалярным типам относятся стандартные типы и типы, определяемые пользователем.

Стандартные типы включают целые, действительные, символьный, логические и адресный типы. Типы, определяемые пользователем - перечисляемый и интервальный.

Структурированные типы имеют четыре разновидности: массивы, множества, записи и файлы.

Кроме перечисленных, PASCAL включает еще два типа - **процедурный и объектный**.

Стандартные типы данных

ЦЕЛЫЕ типы определяют константы, переменные и функции, значения которых реализуются множеством целых чисел, допустимых в данной ЭВМ.

тип	диапазон значений	требуемая память
Shortint	-128 .. 127	1 байт
Integer	-32768 .. 32767	2 байта
Longint	-2147483648 .. 2147483647	4 байта
Byte	0 .. 255	1 байт
Word	0 .. 65535	2 байта

Над целыми операндами можно выполнять следующие арифметические операции: сложение, вычитание, умножение, целочисленное деление, получение остатка от деления. Знаки этих операций:

+ - * **div** **mod**

Результат арифметической операции над целыми операндами есть величина целого типа.

Результат выполнения операции деления (**div**) целых величин есть целая часть частного.

Результат выполнения операции получения остатка от деления (**mod**) - остаток от деления целых. Например:

$$17 \text{ div } 2 = 8, \quad 3 \text{ div } 5 = 0.$$

$$17 \text{ mod } 2 = 1, \quad 3 \text{ mod } 5 = 3.$$

Операции отношения, примененные к целым операндам, дают результат логического типа TRUE или FALSE (истина или ложь).

В языке ПАСКАЛЬ имеются следующие операции отношения: равенство =, неравенство <>, больше или равно >=, меньше или равно <=, больше >, меньше <.

Например – выражение - проверка на четность :

*If (a mod 2) = 0 then b:=c*a
else N:=c+a;*

Если *a* делится на 2 без остатка, то выполняется оператор *b:=c*a*.

Если *a* делится на 2 с остатком, то правая часть выражения (*b:=c*a*) не выполняется, а выполняется оператор, следующее за данным выражением - *N:=c+a*;

К аргументам целого типа применимы следующие стандартные (встроенные) функции, результат выполнения которых имеет целый тип:

Abs(X), Sqr(X), Succ(X), Pred(X),

и которые определяют соответственно абсолютное значение $|X|$, X^2 в квадрате, $X+1$, $X-1$.

Следующая группа стандартных функций для аргумента целого типа дает действительный результат:

Sin(X), Cos(X), ArcTan(X), Ln(X), Exp(X), Sqrt(X).

Эти функции вычисляют синус, косинус и арктангенс угла, заданного в радианах, логарифм натуральный, экспоненту и корень квадратный соответственно.

Результат выполнения функции проверки целой величины на нечетность **Odd(X)** имеет значение истина, если аргумент нечетный, и значение ложь, если аргумент четный:

X=5 Odd(X)=TRUE ,

X=4 Odd(X)=FALSE.

Для быстрой работы с целыми числами определены процедуры:

Inc(X) X:=X+1

Inc(X,N) X:=X+N

Dec(X) X:=X-1

Dec(X,N) X:=X-N

Действительные типы определяют те данные, которые реализуются подмножеством действительных чисел, допустимых в данной ЭВМ.

Тип	Диапазон значений	Количество цифр мантиссы	Требуемая память (байт)
Real	2.9e-39 .. 1.7e+38	11	6
Single	1.5e-45 .. 3.4e+38	7	4
Double	5.0e-324 .. 1.7e+308	15	8
Extended	3.4e-4932 .. 1.1e+4932	19	10

Над действительными операндами можно выполнять следующие арифметические операции, дающие действительный результат:

*Сложение +, вычитание -, умножение *, деление /.*

К величинам действительного типа применимы все операции отношения, дающие булевский результат (*false, true*).

Один из операндов, участвующих в этих операциях, может быть целым.

К действительным аргументам применимы функции, дающие действительный результат:

*Abs(X), Sqr(X), Sin(X), Cos(X), ArcTan(X),
Ln(X), Exp(X), Sqrt(X), Frac(X), Int(X), Pi.*

Функция Frac(X) возвращает дробную часть X, функция Int(X) - целую часть X с отбрасыванием дробной части.

Безаргументная функция Pi возвращает значение числа Пи действительного типа.

К аргументам действительного типа применимы также функции

Trunc(X) и Round(X),

дающие целый результат. Первая из них выделяет целую часть действительного аргумента путем отсечения дробной части, вторая округляет аргумент до ближайшего целого.

Логический тип (*Boolean*) определяет те данные, которые могут принимать логические значения TRUE и FALSE.

К булевским операндам применимы следующие логические операции:

not and or xor.

Логический тип определен таким образом, что FALSE < TRUE. Это позволяет применять к булевским операндам все операции отношения.

Символьный тип (*Char*) определяет упорядоченную совокупность символов, допустимых в данной ЭВМ. Значение символьной переменной или константы - это один символ из допустимого набора.

Символьная константа может записываться в тексте программы следующими способами:

1. как один символ, заключенный в апострофы, например:

'A' 'a' 'Ю' 'ю'

2. с помощью конструкции вида #K, где K - код соответствующего символа, при этом значение K должно находиться в пределах 0..255;

K величинам символьного типа применимы все операции отношения.

Для величин символьного типа определены две функции преобразования

Ord(C) Chr(K).

Первая функция определяет порядковый номер символа C в наборе символов.

Вторая - определяет по порядковому номеру K символ, стоящий на K-том месте в наборе символов. Порядковый номер имеет целый тип.

K аргументам символьного типа применяются функции, которые определяют предыдущий и последующий символы:

Pred(C) Succ(C)

Pred('F') = 'E' Succ('Y') = 'Z'

При отсутствии предыдущего или последующего символов значение соответствующих функций не определено.

Для литер из интервала 'a'..'z' применима функция *UpCase(C)*, которая переводит эти литеры в верхний регистр 'A'..'Z'.

Адресный тип (*Pointer*) - определяет переменные, которые могут содержать значения адресов данных или фрагментов программы. Для хранения адреса требуются два слова (4 байта), одно из них определяет сегмент, второе - смещение.

Для получения значения адреса какой-либо переменной введена унарная операция @.

Тип переменных определяется пользователем в разделе описания переменных. Данный раздел определяет служебное слово - ***var***.

Например:

Var

a,b,c : Real;

Объявлены три переменных ***вещественного типа*** – a, b, c.

В зависимости от того, в каком месте программы описаны переменные, они делятся на две группы – глобальные и локальные переменные.

В оттранслированной (собранной для выполнения) программе, для всех переменных отведено место в памяти, и всем переменным изначально присваиваются нулевые значения.

Константы

определяется пользователем в разделе описания переменных - **var**.

Тип констант в языке ПАСКАЛЬ определяется по их виду:

1. **константы целого типа** - это целые числа, не содержащие десятичной точки. Перед описанием константы пишется служебное слово - **const** :

const

Y : Integer = 4;

2. **константы действительного типа** - действительные числа с плавающей точкой:

const

A : Real = 4.0;

3. **логические константы** - логические значения TRUE и FALSE:

const

Flag : Boolean= true;

4. **символьные константы** - либо строки длиной в один символ.

Любой оператор в программе может быть помечен меткой. В качестве метки используются произвольные целые без знака, содержащие не более четырех цифр, либо имена. **Метка ставится перед оператором и отделяется от него двоеточием.** Все метки, используемые в программе, должны быть перечислены в разделе описания переменных **VAR**, например:

Label 3, 471, 29, Quit;

Комментарии

Кроме описаний и операторов ПАСКАЛЬ - программа может содержать **комментарии**, которые представляют собой произвольную последовательность символов (подсказок или пояснений к программе), расположенных между следующими символами:

```
// строка, расположенная после двух косых воспринимается транслятором
// как комментарий;
{
  C:=a+b;    операторы, заключенные в фигурные скобки воспринимается
  F:=sqrt(c); транслятором как комментарий;
}
(*
```

`C:=a+b;` операторы, заключенные в скобки со звездочкой
воспринимается
`F:=sqrt(c);` транслятором как комментарий;
*)
Закомментированные строки изменяют свой цвет на голубой или зеленый.

ПРЕОБРАЗОВАНИЕ ТИПОВ ДАННЫХ .

При вводе числовой информации в рабочую программу с клавиатуры, как правило используют строчный редактор **TextBox**. Свойство компонента **TextBox** - **Text**, описано как **Text : string**.

Следовательно, в свойстве **TextBox.Text** имеет символьный тип.

При различных вычислениях, ЭВМ оперирует с числовыми данными - целочисленными или вещественными. Следовательно, при вводе числовой информации с клавиатуры и при выводе числовой информации на экран, в компоненты потомков класса **TStrings** необходимо предусматривать преобразование соответствующей информации к нужному типу данных:

- в *символьный* - для их визуализации на экране или принтере;
- в *вещественный* или *целочисленный* - для дальнейшей работы с ними в программе.

Для этого в PascalABC NET имеется достаточно разнообразный набор процедур и функций для выполнения этих задач.

Паскалевские процедуры.

Val (S, A, cod) - процедура, преобразующая строку **S** в переменную **A**.

Переменная **A** может иметь целочисленный или вещественный тип.

В переменную **cod** - имеет **целочисленный тип** - integer, передается код ошибки преобразования. Если переменная **cod = 0** (равна нулю) - преобразование прошло успешно. Если **cod < > 0** - произошла ошибка преобразования строки **S** в переменную **A**.

Данная процедура не отслеживает ошибку преобразования. Рекомендуется выполнить проверку переменной **cod**.

Пример:

```
Var
  x : real;
  n : integer;

begin
  .....
  val(TextBox1.text,x,n);
  if n <> 0 then begin
    ListBox1.items.Add('Ошибка преобразования X');
    exit;
  end;
end;
```

В данном примере, в качестве строки **S** выступает компонент **TextBox.Text**. Свойство **Text** - потомок **TString**, т.е. строка. Данная строка преобразуется в

переменную **X**, описанную в разделе объявлений **Var**, как **real**. Код преобразования получила переменная **n**, описанная в разделе объявлений **Var**, как **integer**.

Далее выполняем проверку корректности преобразования -

```
if n <> 0 then begin
    ListBox1.items.Add('Ошибка преобразования X');
    exit;
end;
```

Если **n** не равно нулю (произошла ошибка) то выдаем сообщение в компонент **ListBox1**:

```
ListBox1.items.Add('Ошибка преобразования X');
```

и безболезненно покидаем процедуру ввода и преобразования данных - **exit**, после чего, сможем выполнить ввод данных повторно.

STR (X: i : j, S) - процедура, преобразующая переменную **X** (вещественного или целочисленного типа) в переменную **S** символьного типа. Параметры **i** и **j** задают формат преобразования: **i** - общее число символов, **j** - число знаков после запятой. При преобразовании целочисленной переменной **X** значение **j** должно быть ноль (**j=0**).

Пример:

```
var
    ss,ss1 : string;
    x : real;
    n : integer;
    .....
    .....
    STR(x:10:3,ss);//преобразовали вещественный X в строку SS
    STR(n:6:0,ss1);// преобразовали целочисленное n в строку SS1
```

Функции преобразования Delphi. (совместимы с *PascalABC NET*)

StrToFloat - функция, конвертирует числовую строку типа **string**, например '123.456', в значение с плавающей точкой с типом **Extended**.

Она поддерживает целый формат, формат с плавающей точкой и научный (экспоненциальный) формат.

Пример

```
var
    ss : string='123.456';
    x : Extended.
    .....
```

.....

X := strtfloat(ss);// преобразовали строку SS в переменную X

FloatToStrF - Функция, конвертирует значение переменной вещественного типа в строку (String), с большим управлением по форматированию через значения: **Format**, **n** - *общее количество символов* и **m** - *количество знаков дробной части*.

<переменная типа string>:=**floattostr**(<переменная вещественного типа >, <формат>, <n>, <m>)

Параметр **Format** определен в **TFloatFormat** (SysUtils) типом:

ffCurrency	: \$2,345.60	// денежный формат
ffExponent	: 2.3456E+04	// научный формат
ffFixed	: 2345.60	// число с фиксированной точкой
ffGeneral	: 2345.6	// число с фиксированной точкой 1
ffNumber	: 2,345.6	// число с фиксированной точкой 2

Примеры преобразований:

var

x : Extended = 1234.567;

SS : string;

.....
.....

btgin

// денежный формат:

ss:=Floattostr(x,ffCurrency, 8, 4)

// ss получит значение = \$ 1,234.5670

ss:=Floattostr(x,ffCurrency, 4, 4)

// ss получит значение = \$ 1,234.0000

ss:=Floattostr(x,ffCurrency, 4, 2)

// ss получит значение = \$ 1,234.00

ss:=Floattostr(x,ffCurrency, 2, 4)

// ss получит значение = 1,2e+0003

// научный формат

ss:=Floattostr(x, ffExponent, 8, 4)

```
// ss получит значение = 1.2345670E+0003
ss:=Floattostr(x, ffExponent, 4, 4)
// ss получит значение = 1.235E+0003
ss:=Floattostr(x, ffExponent, 4, 2)
// ss получит значение = 1.235E+03
ss:=Floattostr(x, ffExponent, 2, 4)
// ss получит значение = 1.2E+0003
```

// формат с фиксированной точкой

```
ss:=Floattostr(x, ffFixed, 8, 4)
// ss получит значение = 1234.5670
ss:=Floattostr(x, ffFixed, 4, 4)
// ss получит значение = 1234.0000
ss:=Floattostr(x, ffFixed, 4, 2)
// ss получит значение = 1234.00
ss:=Floattostr(x, ffFixed, 2, 4)
// ss получит значение = 1.2E+0003
.....
.....
end;
```

FloatToStr - Функция, конвертирует значение переменной вещественного типа в строку (String) без форматирования. Если с числом производились какие либо математические действия, то в качестве дробной части данная функция выдаст 15 знаков.

IntToStr - функция, преобразует целое число **Integer** или **Int64** в строку.

Пример:

```
var
  x : integer = 1234;
  SS : string;
.....
.....

begin
  ss:=inttostr(x); //ss получит значение '1234'
end;
```

StrToInt - функция, преобразует строку **Integer** или **Int64** в целое число .

Пример:

var

x : integer;

SS : string = '1234';

.....
.....

begin

x:=strtoint(ss); //x получит значение = 1234

end;

Процедуры и функции.

Алгоритм решения задачи проектируется путем декомпозиции всей задачи в отдельные подзадачи. Обычно подзадачи реализуются в виде подпрограмм.

Подпрограмма - это последовательность операторов, которые определены и записаны только в одном месте программы, однако их можно вызвать для выполнения из одной или нескольких точек программы. Каждая подпрограмма определяется уникальным именем.

В языке ПАСКАЛЬ существуют два типа подпрограмм - процедуры и функции.

Процедура и функция - это именованная последовательность описаний и операторов.

При использовании процедур или функций ПАСКАЛЬ - программа должна содержать текст (тело) процедуры или функции и обращение к процедуре или функции.

Во многих задачах необходимо передавать имена процедур и функций в качестве параметров. Для этого в **PASCAL** введен **новый тип данных - процедурный или функциональный**, в зависимости от того, что описывается. Описание процедурных и функциональных типов производится в разделе описания типов:

type

Function M1(z: Real): Real;

Procedure V_2(a,b : Real);

Или в разделе описания переменных:

Var

Function M1(z: Real): Real;

Procedure V_2 (a,b: Real);

Функциональный и процедурный тип определяется как заголовок процедуры или функции со списком **формальных параметров** - (z: Real): Real; (a,b: Real);

После объявления переменных процедурного типа, **необходимо написать** в исполняемой части программы (**implementation**) **реальные процедуры или функции**, к именам которых будут обращаться.

Например:

Procedure V1(a,d: real);

Begin

Оператор 1

.....

Оператор N

End;

Для того, чтобы обратиться к данной процедуре, необходимо, в качестве оператора в программе, задать ее имя со списком *фактических параметров*. Например:

Begin

Оператор 1

.....

Оператор N

V1(x,y); // обращение к процедуре

End;

Здесь **XУ** – **Фактические параметры**, которые заместят **формальные параметры в процедуре Procedure V1(a,d: real);**

Процедура может содержать такие-же разделы описаний, что и ПАСКАЛЬ - программа, а именно: разделы описания модулей, меток, констант, типов, переменных, процедур и функций.

Базовые конструкции языка

Структурированная программа (или подпрограмма) - *это программа, составленная из фиксированного множества базовых конструкций*. Рассмотрим основные определения и способы образования этих конструкций в схемах алгоритмов.

Из операций, развилок и слияний строятся базовые конструкции: *следование, ветвление, цикл*. Применяя только эти три конструкции, можно реализовать алгоритм решения любой задачи.

- ☐ *Конструкция, представляющая собой последовательное выполнение двух или более операций, называется следованием.*
- ☐ *Конструкция, состоящая из развилки, двух операций и слияния, называется ветвлением. Одна из операций может отсутствовать.*
- ☐ *Конструкция, имеющая линии управления, ведущие к предыдущим операциям или развилкам, называется циклом.*

В языке ПАСКАЛЬ шесть базовых конструкций, это:

- *следование;*
- *ветвление;*
- *цикл с предусловием;*
- *цикл с постусловием;*

- *цикл с параметром;*
- *вариант.*

Условный оператор

if < булевское выражение> **then** оператор_1
else оператор_2;
 оператор_3;

Перед **else**, после оператора_1, точка с запятой не ставится !!!

Условный оператор в короткой форме работает по правилу: если булевское выражение истинно, то выполняется оператор_1, далее выполняется оператор_3, следующий за условным оператором. Если булевское выражение ложно, то будет выполняться оператор_2, следующий за этим условным оператором.

Пример 1 (простое сравнение):

IF $a > b$ **then** $c := a - b$
Else $c := a + b$;
 $D := D + c$;

В данном примере сравниваются две величины **a** и **b**. Если $a > b$ истина (булевское выражение **true**), будет выполняться оператор $c := a - b$, а затем $D := D + c$;

Если $a > b$ ложь (булевское выражение **false**), будет выполняться оператор $c := a + b$, а затем $D := D + c$;

Пример 2 (сравнение с логическим **и** - **and**):

IF $(dx > 0)$ and $(dy < 0)$ **then** showMessage('четвертая четверть')
Else showMessage('это не четвертая четверть');

В данном примере сравнивается результат проверки двух выражений $(dx > 0)$ и $(dy < 0)$. **Если оба выражения соответствуют истине**, то на экране появится сообщение - 'четвертая четверть'.

Если результат проверки одного или обоих выражений ложь, на экране появится сообщение - 'это не четвертая четверть'

В данном примере **ShowMessage** - стандартная функция динамической библиотеки Dialogs дельфи, которая выводит на экран панель с сообщением, которое заключено в скобки.

Пример 3 (сравнение с логическим **или** - **OR**):

IF $(dx > 0)$ or $(dy < 0)$ **then** showMessage('условие выполнено')

Else showmessage('это условие не выполнено');

В данном примере сравнивается результат проверки двух выражений ($dx > 0$) или ($dy < 0$). *Если одно из выражений соответствует истине*, то на экране появится сообщение - 'условие выполнено'.

Если результат проверки обоих выражений ложь, на экране появится сообщение - 'это условие не выполнено'

Цикл с предусловием

while <выражение> **do** оператор;

Если операторов несколько, то они заключаются в конструкцию **begin** (начало) - **end** (конец).

```
while <выражение> do begin  
    оператор_1;  
    оператор_2;  
end; // конец цикла  
    оператор_3;
```

Данная конструкция предполагает сначала проверку условия, а затем выполнение операторов. Если, при обращении к циклу, проверка условия будет ложь (**false**), то будет выполняться оператор, следующий за окончанием цикла - оператор_3;

Пример (чтение текстового файла с диска):

```
i:=0;
```

```
While not eof(t) do begin
```

```
    Readln(t, x,y);
```

```
    X1[i]:=x;
```

```
    Y1[i]:=y;
```

```
    i:=i+1;
```

```
End;
```

В данном примере логическим условием является выражение **while not eof(t)**, которое означает "пока нет конца файла **t**, выполнить (**do begin**) операторы, стоящие перед **End;** после чего вернуться на начало цикла **while not eof(t)**. Чтение будет продолжаться до тех пор, пока не будет достигнут конец файла **eof(t)**, т.е. выражение **not eof(t)** станет ложью – **false**.

// **eof** - признак конца файла, прописывается автоматически в конце текстового файла, аббревиатура от **end of file** - конец файла. **t** - файловая переменная, которая содержит имя рабочего файла и путь к нему. **Readln** -

оператор паскаля, читающий строку файла до конца, с "переводом каретки" на новую строку. //

Цикл с постусловием

repeat
 оператор;
until <выражение >;

В отличие от цикла с предусловием, представленная конструкция выполнится хотя бы один раз. Проверка условия выполняется после выполнения оператора.

Цикл с параметром

For *i* := *параметр_1* **to** *параметр_2* **do** *оператор*;
For *i* := *параметр_1* **downto** *параметр_2* **do** *оператор*;

Если в цикле несколько операторов, то они заключаются в конструкцию
begin ... end;

For *i* := *параметр_1* **to** *параметр_2* **do**
begin
 оператор_1;
 оператор_2;
end;

Здесь *i* - переменная цикла, которая вначале работы принимает значение *параметр_1*. После отработки тела цикла (*оператор_1*; *оператор_2*;) *i* - переменная цикла, принимает значение *i:=i+1*(если после *параметра_1* стоит *to*) , т.е. увеличивается на единицу.

Если стоит **downto**, то переменная цикла, принимает значение *i:=i-1* после чего выполняется неявная проверка на превышение *параметра_2* – **if** *i* > *параметра_2* то цикл завершается(если после *параметра_1* стоит *to*) и наоборот, **if** *i* < *параметра_2* то цикл завершается - если стоит **downto**.

Пример: Найти факториал числа 10.

P_f:=1;
For *j:=1 to 10 do P_f:= P_f * j;*

В ПАСКАЛЬ определена стандартная процедура **Break**, которая выполняет безусловный выход из цикла, даже если *параметр_2* не достигнут.

Пример 2 Определить номер элемента массива **mas**, состоящего из 100 элементов целого типа, величина которого превышает заданное значение **max_1**.

```
For j:=1 to 100 do begin
  If mas[j]>max_1 then begin
    ListBox1.items.Add (inttostr(j));
    Break;
  End; // if
End; // for
```

inttostr(j) - функция - преобразует целочисленную переменную **j** в строку.

Оператор варианта

```
case <выражение> of
  если выражение _1 : оператор_1;
  если выражение _2 : оператор_2;
  ...
  если выражение _N : оператор_N
else
  оператор_N+1;
end;
```

Данный оператор позволяет программисту, *в зависимости от условия*, выбирать вариант действия в программе.

Рассмотрим пример работы программы «калькулятор» При нажатии на кнопки действия **+**, **-**, ***** и т.д. необходимо выбрать соответствующие операции с исходными данными.

Предположим, нажали на кнопку «*****» - умножить. При этом, “условие” получило значение=3.

```
N:=3;
.....
.....
Case N of
  1: c:=a+b;
  2: c:=a-b;
  3: c:=a*b;
  4: c:=a/b;
End
```

Else

ListBox1.items.Add (***‘Такого условия нет’***);

End;

В нашем примере N=3, выполнится оператор с вариантом 3: c:=a*b; Если вариант отсутствует, выполнится оператор стоящий после Else -

ListBox1.items.Add (***‘Такого условия нет’***);

Структурные типы данных

К структурным типам данных относятся: массивы , записи и множества.

Массивы данных

Массив — набор однотипных данных, расположенных в памяти, доступ к которым осуществляется по индексу (индексам). В самом простом случае массив имеет фиксированное количество элементов (статический массив), но бывают и динамическиемассивы, меняющие свою размерность в ходе выполнения задачи.

Статические массивы.

Для того, чтобы объявить статический массив, необходимо в разделе описания переменных - **VAR** - сделать следующее объявление:

```
<имя массива> : array [<индекс первого элемента> .. <индекс последнего элемента>] of  
<тип данных>
```

Пример: Зададим массив данных с именем Mass, длиной в 11 элементов вещественного типа.

var

```
mass : array[0..10] of real;
```

Данным объявлением мы выделили память под 11 переменных вещественного типа. Каждая переменная - значение массива, выбирается по ее индексу - порядковому номеру данных в массиве.

Предположим, что элементами массива являются следующие числа:

2.5 7.0 3.1 9.8 3.3 5.6 7.2 4.3 6.9 8.7 9.9 10.2

Но каждому числу в массиве соответствует свой индекс:

mass												
индекс	0	1	2	3	4	5	6	7	8	9	10	11
значение	2.5	7.0	3.1	9.8	3.3	5.6	7.2	4.3	6.9	8.7	9.9	10.2

Тогда, чтобы получить соответствующие данные из массива, необходимо к имени массива добавить (в квадратных скобках) индекс элемента.

Например, следующим элементам массива соответствуют данные:

```
mass[0] -> 2.5;   mass[7] -> 4.3;   mass[10] -> 9.9
```

Рассмотренный выше массив называется *одномерным массивом*.

Но массивы могут быть и *многомерными*:

Например, запись следующего типа задает двумерный массив:

var

mass : array[0..10, 0..100] of real;

Массив описанный таким образом содержит 11 строк по 101 элементу (столбцов) данных вещественного типа в каждой строке

Объявим массив, состоящий из двух строк и 11 столбцов

var

mass : array[0..1, 0..10] of real;

Предположим, что данному массиву соответствуют следующие числа:

Строка 0 2.5 7.0 3.1 9.8 3.3 5.6 7.2 4.3 6.9 8.7 9.9 10.2

Строка 1 1.9 3.2 5.1 7.4 8.8 3.9 2.1 2.9 6.7 4.2 5.5 7.8

mass												
столбец	0	1	2	3	4	5	6	7	8	9	10	11
Строка 0	2.5	7.0	3.1	9.8	3.3	5.6	7.2	4.3	6.9	8.7	9.9	10.2
Строка 1	1.9	3.2	5.1	7.4	8.8	3.9	2.1	2.9	6.7	4.2	5.5	7.8

Элементы массива в памяти ЭВМ следуют друг за другом так, что при переходе от младших адресов к старшим, наиболее быстро меняется самый правый индекс массива. Таким образом, в памяти ЭВМ будет записана сначала строка [0], а следом за ней - строка [1]

Для того, чтобы получить соответствующий элемент данного массива, необходимо *сначала указать номер строки*, а затем, через запятую, *номер столбца*.

Например, следующим элементам массива соответствуют данные:

mass[0,5] -> 5.6; mass[1,7] -> 2.9; mass[0,10] -> 9.9

Работа со статическими массивами не всегда удобна, поскольку количество данных в различных задачах может превышать размер памяти отведенной под статический массив. А это может привести к исключительной ситуации в процессе выполнения программы. Программа может "повиснуть". С другой стороны есть и плюс у статического массива. Программист не задумывается об освобождении выделенной памяти под массив в "общую кучу" памяти компьютера.

Динамические массивы.

В 1998 году разработчики Delphi реализовали достаточно гибкий механизм для работы с динамическими массивами. Динамические массивы не имеют фиксированного размера или длины. Для объявления такого массива достаточно записать:

```
Var  
    mass : array of <тип данных>;
```

Например:

```
mass : array of real;
```

При таком объявлении массива память не выделяется. Для выделения памяти динамическому массиву, используется процедура **SetLength**:

Например:

```
SetLength(mass,1);
```

После вызова этой процедуры будет выделена память для 1 элемента вещественного типа данных массива, индекс которого будет mass[0] (*обратите внимание: индексирование начинается с нуля, а не с единицы!*).

Стандартная функция **Length** возвращает длину массива (количество элементов массива).

```
Length(mass) = 1;
```

Рассмотрим пример работы с одномерным динамическим массивом.

В данном примере мы будем читать исходные данные из текстового файла в динамический массив, с наращиванием его длины. (*Более подробно, работа с файловой системой будет рассмотрена в следующем разделе*)

```
unit Unit1;
```

```
interface
```

```
.....
```

```
.....
```

```
Var
```

```
    x : array of real;
```

```
    t : textfile;
```

```
    Procedure read_text; // объявили переменную процедурного типа
```

```
.....
```

```
.....
```

```
implementation
```

```
Procedure read_text; // реализация переменной процедурного типа
```

```
begin
```

```
    SetLength(x,1);
```

```
    i:=0;
```

```

    while not eof(t) do begin
        if length(x)<=i then setlength(x, length(x)+1);
        readln(t, x[i]);
        inc(i);
    end; //while
    closefile(t);
end; // Procedure read_text
.....
.....
IF OpenFileDialog1.ShowDialog =
system.Windows.Forms.DialogResult.OK then begin
    assignfile(t, openFileDialog1.filename);
    reset(t);
    read_text; // обращение к процедуре чтения
end;
.....
.....
end.

```

Рассмотрим подробно данный пример.

Имеется интерактивный Редактор Кода - unit1.

В области интерфейсной части, в разделе описания переменных **VAR** объявлен динамический массив **x : array of real;** кроме этого объявлена переменная **t : textfile** для работы с текстовым файлом и объявлена переменная процедурного типа - **Procedure read_text** - без формальных параметров.

В разделе **implementation** - исполняемой части Редактора Кода - описано тело переменной процедурного типа - **Procedure read_text;**

При необходимости прочитать информацию из текстового файла, мы используя компонент - OpenFileDialog1, палитры компонентов Диалоговые окна;

IF OpenFileDialog1.ShowDialog = system.Windows.Forms.DialogResult.OK then begin

Данный оператор открывает модальное окно для работы с файлами и папками (в просторечьи - проводник).

, выбираем и открываем файл с исходными данными -

assignfile(t, openFileDialog1.filename).

Свойство компонента **openFileDialog1** - **filename** - содержит адрес и имя выбранного файла. Свойство filename передается в файловую переменную **t**. В дальнейшем **t** является источником информации.

Функция **reset(t)** открывает файл для последовательного чтения сначала.

Затем обращаемся к процедуре **Procedure read_text** *no ee имени* - **read_text.**

Далее в процедуре:

1. Устанавливаем минимальную длину динамического массива `SetLength(x,1);`
2. Поскольку заранее неизвестно количество данных в файле, для его чтения используем цикл с предусловием **while not eof(t) do begin**. Перед циклом задаем значение счетчика цикла (количества прочитанной информации) - `i:=0;`
3. Перед чтением файла проверяем длину динамического массива
if length(x)<=i then setlength(x, length(x)+1);
если условие = true то увеличиваем длину массива, как текущую длину `length(x)` и к ней добавляем 1 - `setlength(x, length(x)+1);`
4. После этого читаем данные из текстового файла в динамический массив - `readln(t, x[i]);` .
Индексом данных массива является переменная счетчика цикла `i`.
5. Затем увеличиваем переменную цикла на 1 - `inc(i);` и переходим на пункт 2 - **while not eof(t) do begin**. Если условие цикла дает true, то цикл продолжается далее, если false - цикл завершается, текстовый файл закрывается - `closefile(t)`. Работа программы переходит к оператору, стоящему после оператора вызова процедуры чтения - `read_text;`

*Для того, чтобы обрезать (укоротить) массив, используют процедуру **setlength**.*

Предположим, длина динамического массива `x=10`, используя его повторно, нам потребуется всего 5 элементов. Установим новую длину: `setlength(x,5);`

Обрезанные элементы массива автоматически обнулятся и вернутся в общую кучу памяти.

Важно !!! При завершении работы программы На программиста возлагается задача освободить память, выделенную под динамический массив. Делает это функция **Finalize** или **nil**

Пример для массива X :

Finalize(x); // очистили память, занятую под массив X.

или

X:=nil; // очистили память, занятую под массив X.

Мы рассмотрели принцип работы с одномерными динамическими массивами. Но, как и статические, динамические массивы могут быть многомерными.

Многомерные динамические массивы.

Для объявления многомерного динамического массива используют следующие объявления:

Var

X : array of array of real; - объявлен двумерный массив вещественного типа;

Y : array of array of array of real; - объявлен трехмерный массив вещественного типа;

Рассмотрим принцип работы многомерного динамического массива на *примере двумерного*.

Var

X : array of array of real;

При данном объявлении, как говорилось выше, память не выделяется.

Для задания количества строк и столбцов массива используют известную нам функцию **SetLength**;

(Под столбцами будем понимать длину строки)

Для данного массива **X** установим: количество строк -2, столбцов 11.

1. Устанавливаем количество строк:

SetLength(X,2);

2. Устанавливаем количество столбцов (напоминаем, что счет строк и столбцов начинается с нуля):

setlength(x[0],10);

setlength(x[1],10);

В динамическом многомерном массиве длина строк может различаться:

setlength(x[0],10);

setlength(x[1],5);

В данном примере нулевая строка содержит 11 элементов, а первая строка - 6.

К элементам массива обращаются так же, как и в статическом массиве.

Освобождение памяти выполняется так же, как и в одномерном динамическом массиве.

Пример работы с двумерным динамическим массивом.

В данном примере мы будем читать исходные данные из нескольких текстовых файлов в динамический массив, с наращиванием длины его строк.

Каждая строка соответствует данным отдельного файла.

.....

var

x : array of array of real;

i, j : word;

t : textfile;

.....

.....

```
i:=0;
while i<=ListBox1.Items.Count-1 do begin
  assignfile(t,ListBox1.Items[i]);
  reset(t);
  if length(x)<=i then setlength(x,length(x)+1); // Добавляем строку
  j:=0;
  While not eof(t) do begin
    if length(x[i])<=j then setlength(x[i],length(x[i])+1); { Увеличиваем длину
строки }
    readln(t, x[i,j]);
    inc(j);
  end; //while not eof(t)
  closefile(t);
  inc(i);
end; //while i<=ListBox1
```

Здесь:

ListBox1.Items.Count-1 - количество имен файлов занесенных в список
ListBox1.

ListBox1.Items[i] - имя файла, подлежащего обработке.

Записи.

Записи - очень важный и удобный инструмент. Даже не применяя специальные технологии, с помощью записей можно создавать собственные **базы данных**. Записи - это структура данных, каждый элемент которой имеет собственное имя и тип данных. Элемент записи иначе называют *поле*.

Описание записи имеет вид:

```
type
  <тип_имени_записи> = record
    <название_поля> : тип_данных ;
    . . .
    <название_поля> : тип_данных ;
end;
.....
var
  <имя записи> : <тип_имени_записи>;
```

Названия полей, имеющих одинаковый тип, можно, как и в случае описания простых переменных, указывать в одну строку через запятую. Для обращения к полю записи сначала указывают имя записи, затем точку, затем имя поля:

```
<имя записи>.<название_поля>;
```

Записи создают в разделе описания типов.

Например:

```
type
  M_cod = record    // тип_имени_записи
    data_set: string[20];
    adr : string[100]
    obgect : string[50];
    Tip_obg : string[30];
    n_obg : string[10];
    n_swid : string[10];
    wlad : string[50];
end; // type
.....
var
  cod : M_cod; // <имя записи> : <тип_имени_записи>;
  index_file : file of M_cod;
  ss : string;
```

В данном примере создана запись для обследования каких либо объектов и типизированный файл записей - index_file.

Сама запись состоит из 7 полей:

1. data_set - Данному полю отводится 20 символов - дата обследования.
2. adr - 100 символов - адрес объекта.
3. obgect - 50 символов - название объекта.
4. Tip_obg - 30 символов - тип объекта.
5. n_obg - номер объекта по кадастровому свидетельству.
6. n_swid - 10 символов - номер кадастрового свидетельства
7. wlad - 50 символов - данные о владельце.

*В данном случае поля записи перечислять через запятую нельзя, поскольку длины полей типа **string** разные.*

Получить значения поля записи:

<имя записи>.<название_поля>;

Например - получить данные о владельце:

SS:= cod.wlad;

Таких записей может быть достаточно много и их можно хранить в типизированном файле. (каждый отдельный элемент такого файла является записью с именем COD).

Множества.

Множества -это тип данных, представляющий собой набор взаимосвязанных по какому-либо признаку или группе признаков объектов, которые можно рассматривать как единое целое. Количество элементов, входящих в множество, может меняться от 0 до 256. Каждый объект в множестве называется элементом множества. Если множество не имеет элементов, то оно называется пустым.

Описание множества имеет тип

<идентификатор> = **set of** <базовый тип>

Где базовый тип - тип элементов данных. Обычно он задается диапазоном.

Пример задания множества:

var

dec1 : **set of** 0..9; //Множество задается диапазоном

dec2 : **set of** byte; //Можно указать тип данных, если его размер - 1 байт

Над множествами определены следующие операции:

Пусть M1=[1,2,3,5,8], M2=[1,2,4,6,8]

- Пересечение множеств (*). Результат содержит элементы общие для обеих множеств ($M1 * M2 = [1, 2, 8]$)
- Объединение множеств (+). Результат содержит элементы первого и второго множеств ($M1 + M2 = [1, 2, 3, 4, 5, 6, 8]$)
- Разность множеств (-). Результат содержит элементы первого множества, не входящие во второе ($M1 - M2 = [3, 5]$)
- Проверка эквивалентности (=). Возвращает True если множества эквивалентны (все элементы совпадают).
- Проверка неэквивалентности (<>). Возвращает True если множества неэквивалентны.
- Проверка вхождения (<=). Возвращает True, если первое множество включено во второе.
- Проверка принадлежности (in). Возвращает True если выражение имеет значение, принадлежащее множеству ($2 \text{ in } M1 = \text{true}$).

Также существует возможность добавлять или исключать из множества элементы:

```
...
M1:=M1-[2]; //Теперь M1=[1,3,5,8]
M2:=M2+[12]; //Теперь M2=[1,2,4,6,8,12]
```

Множества можно использовать для проверки, входит ли какое-нибудь число или буква в некий диапазон множества.

Следующий пример демонстрирует ввод в редактор Edit1 только чисел, а в качестве разделителя - точку. Все остальные символы игнорируются.

```
.....
implementation
var
    m1 : boolean = false;
    {$R *.dfm}

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    if (key='.') and (length(edit1.Text)=0) then key:=#0;
    if (key='.') and (m1=false) then begin
        m1:=true;
        exit;
    end
    else if not(key in ['0'..'9']) then key:=#0
end;

end.
```

В данной процедуре, при вводе информации в редактор edit1, отслеживается событие **On KeyPress**. Данная процедура возвращает (символ) нажатой клавиши - Key типа Char (например запись '.' - соответствует нажатой клавише *точке*).

В строке

```
if (key='.') and (length(edit1.Text)=0) then key:= #0;
```

производится проверка установки разделителя - точки - в первой позиции.

Если длина length текстовой части редактора (length(edit1.Text)=0) равна нулю, то

символу переменной key присваивается код **#0 - отсутствие символа** (точку не ставим). Положение курсора в редакторе не меняется.

В следующем выражении отслеживается установка разделителя - точки.

```
if (key='.') and (m1=false) then begin  
  m1:=true;  
  exit;  
end  
else if not(key in ['0'..'9']) then key:= #0;
```

Если точка установлена в первый раз, то переменной m1 присваивается true

и мы покидаем процедуру - exit; - до следующего ввода символа,

иначе - else - производится проверка - **if not**(key in ['0'..'9']) **then** key:= #0; - входит ли введенный символ в данное множество.

Если нет, то символу переменной key присваивается код #0 - отсутствие символа, если да, то введенный символ появляется в поле редактора.

Файлы и файловая система.

Общие понятия.

Файл — это определенное количество информации (программа или данные), имеющее имя и хранящееся в долговременной (внешней) памяти.

Имя файла состоит из двух частей, разделенных точкой: собственно имя файла и расширение, определяющее его тип (программа, данные и т. д.). Имя файлу и его расширение задает пользователь. Расширение файла — часть имени файла, отделённая самой правой точкой в имени. Как правило, расширение файла несет смысловую нагрузку о типе данных и способе хранения информации.

Например:

- .txt - расширение .txt присваивается текстовым файлам;
- .bmp, .tif - содержат изображения в соответствующих форматах;
- .exe - откомпилированные (исполняемые) программные модули;
- .ini - файлы инициализации и т.д.

Имена и расширения файлов могут прописываться как прописными, так и строчными символами.

В имени файла разрешается использовать национальные символы, пробелы и другие символы, за исключением следующих девяти: $\backslash$: * ? " < > | .

Файловая система - это система хранения файлов и организация каталогов. Она определяет формат физического хранения информации, которую принято группировать в виде папок и файлов. Конкретная файловая система определяет размер имени файла, максимальный возможный размер файла, набор атрибутов файла.

Самым распространённым видом файловой системы являются иерархические ФС, обеспечивающие хранение файлов в древовидной структуре каталогов.

Стандартные компоненты PascalABC NET для работы с файловой системой.

Для работы с файлами в палитре компонентов предусмотрен раздел 'Диалоговые окна' (рис.18)

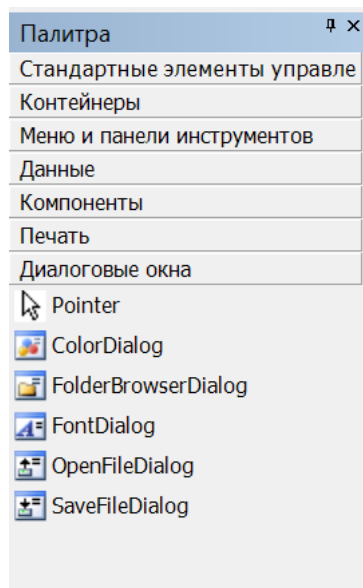
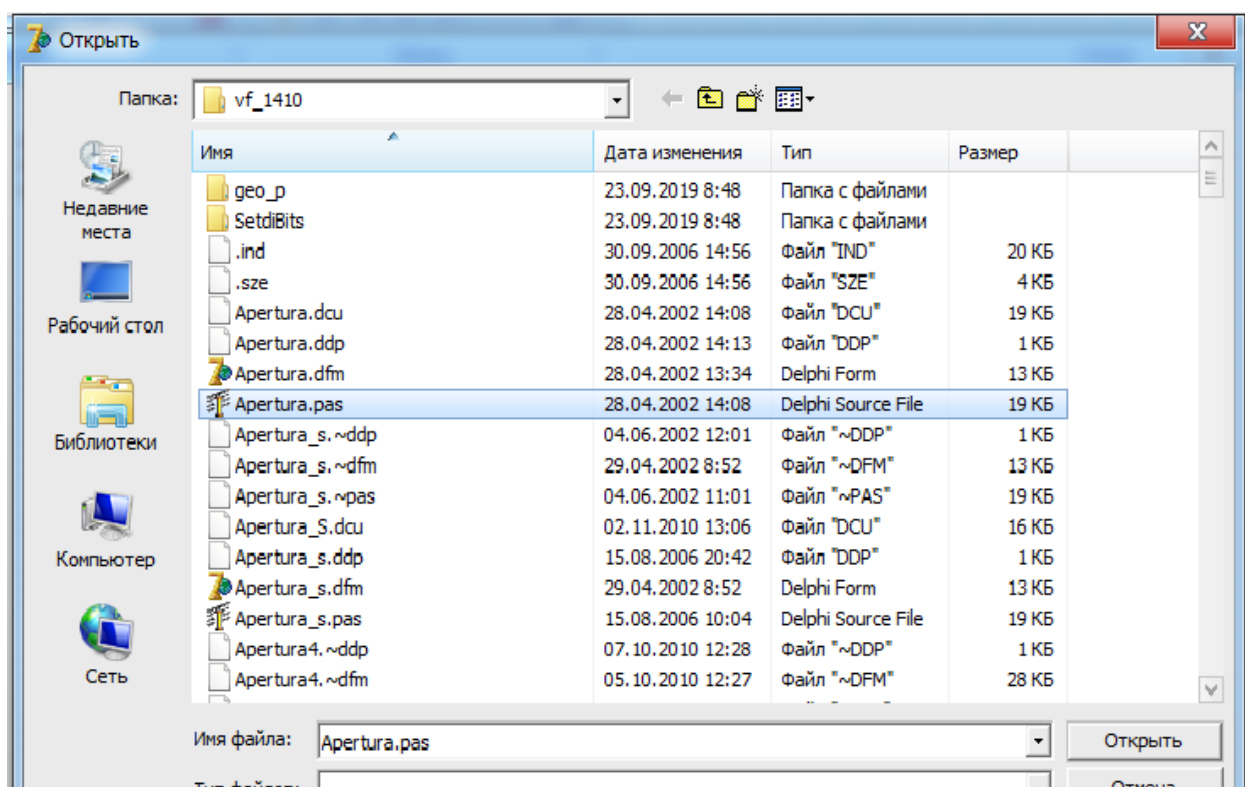


Рис. 18

которая содержит 5 **не визуальных** компонентов для работы с файлами и папками (рис. 18)

Рассмотрим основные компоненты.

1. OpenFileDialog - открывает стандартное диалоговое окно для работы с файлами и папками (рис.14).



Для того, чтобы воспользоваться данным компонентом, необходимо создать обработчик события, активирующий компонент OpenFileDialog. Таким событием может быть щелчок мышки по кнопке "Открыть файл".

Для того, чтобы создать такое событие, необходимо в панели **Инспектор объектов**

для кнопки "Открыть файл" выбрать закладку */событие/*, в которой выбрать событие **Click** (щелчок мышки по кнопке) - дважды щелкнув мышкой **по пустому полю** справа от свойства. В разработчике кода появится пустая процедура обработки щелчка по кнопке:

```
procedure TForm1.Button1_Click(Sender: TObject);
begin
```

```
end;
```

в которой нужно записать **стандартный код активации компонента OpenFileDialog1:**

```
procedure TForm1.Button1_Click(Sender: TObject);
begin
  IF OpenFileDialog1.ShowDialog = system.Windows.Forms.DialogResult.OK
  then begin
    // здесь пишутся необходимые операторы
  end;

end;
```

Адрес и имя выбранного файла передаются в свойство компонента

OpenFileDialog1 - FileName.

Например, получить имя выбранного файла: **ss:=OpenFileDialog1.FileName;**
переменная SS должна быть описана как **string;**

2. компонент **SaveFileDialog** - служит для сохранения файла. Его работа аналогична с компонентом **OpenFileDialog**.

РАБОТА С ФАЙЛАМИ.

При разработке прикладных задач, практически всегда требуется чтение - сохранение данных на носители памяти. Данные задачи обеспечивает работа с файлами и с файловой системой.

По типу хранения данных файлы делятся на:

- текстовые;
- типизированные;
- Не типизированные (двоичные);

Для большинства разрабатываемых задач, чаще всего используют текстовые файлы.

Текстовые файлы

По своей структуре текстовый файл содержит разные или равные по длине строки, отделенные символами #13#10 (eoln - аббревиатура от end of lines - конец строки), записанных на носителе памяти последовательно друг за другом.

Конец файла имеет признак eof - аббревиатура от end of file - конец файла.

Есть 2 основных метода работы с текстовыми файлами - через файловые потоки и паскалевский способ.

Файловые потоки.

У всех потомков класса TStrings (ListBox.Lines, Listbox1.Items ...) есть методы записи и чтения в файл - SaveToFile, LoadFromFile. Преимущество - простота использования и довольно высокая скорость, недостаток - читать и сохранить файл можно только целиком.

Примеры:

1. Загрузка текста из файла в компонент Memo и Listbox:

```
Listbox1.Items.loadfromfile(OpenDialog1.FileName);
```

2. Сохранение в файл:

```
Listbox1.Items.SaveToFile(SaveDialog1.FileName);
```

Паскалевский метод доступа к текстовому файлу.

Для более тонких операций над текстовыми файлами используют паскалевский метод доступа.

1. Для доступа к текстовым файлам используется переменная типа **TextFile**, которую объявляют в разделе описания переменных:

Var

```
t : textfile;
```

```
f : textfile;
```

2. При открытии файла для работы, в данную переменную передается адрес и имя выбранного файла с помощью функции **AssignFile**:

с использованием компонента OpenFileDialog :

```
AssignFile(t, OpenFileDialog1.FileName);
```

или указав абсолютный адрес файла:

```
AssignFile(t, 'c\my_dat\program.ini');
```

Здесь:

C - расположение файла на диске;

my_dat - папка, в которой находится файл;

program.ini - имя файла.

В дальнейшем, данная переменная **t** служит источником информации при работе с файлом.

3. Затем, для определенного вида работ, необходимо открыть файл. Для этого имеется три функции:

- **Rewrite(t)** - создает новый (пустой) файл или уничтожает старый (без предупреждения) и создает новый пустой файл для последовательных записей;
- **Reset(t)** - открыть файл для чтения-записи (сначала):
- **Append(t)** - файл есть и его надо открыть для дописывания строк в конец файла:

4. В конце работы открытый файл нужно закрыть: **CloseFile(t);**

Иначе часть информации может быть потеряна.

Чтение предварительно открытого файла/

ReadLn(t, s) - будет прочитана вся текущая строка и позиция чтения переведена на следующую строку. S описана как string;

ReadLn(t,a,b) - будет прочитаны переменные a, b, *автоматически произойдет преобразование типа данных string к тому типу данных, каким описаны переменные a, b* и позиция чтения будет переведена на следующую строку.

При таком чтении *переменные a,b должны иметь один и тот же тип данных. Например: var a, b : real;*

Не допускается чтение разнотипных данных. например
a : string; b : real;

Допускается последовательное чтение переменных отличных от описания типа string:

Read(t, a) - позиция чтения не будет переведена на следующую строку, а остановится перед следующей переменной b, которую затем также можно прочитать - **Read(t, b)**. Для перехода на следующую строку служит пустой оператор чтения - **Readln(t);**

Запись в текстовый файл.

Запись целой строки: **Writeln(t,s);**

Запись кусочка строки (т.е. следующая операция записи будет произведена в ту же строку): **Write(t,s);**

Если переменная S содержит больше 255 символов (т.е. является длинной строкой), то таким способом она не запишется, в файл вместо строки попадут 4 байта указателя на нее. Надо делать так:

Writeln(t,pointer(s)^);

где **pointer(s)^** - указатель на строку (адрес строки).

При сохранении данных из рабочей программы в текстовый файл, все переменные необходимо преобразовать к типу string. (смотри раздел - *Преобразование типов данных*)

В ПАСКАЛЕ, при сохранении данных в текстовый файл, допускается следующий вид преобразования переменных, отличных от типа string:

Writeln(t, a:10:3, b:12:3);

Здесь:

a, b - описаны в программе типом **real**;

Конструкция

a:10:3 - преобразует вещественный тип данных в символьный. Цифра 10 - означает, что под число отводится 10 символов (каждый символ - 1 байт), цифра 3 - сколько знаков показать в дробной части.

Тоже самое для переменной **b** - 12 символов, 3 знака после запятой.

Пример (описан в разделе *Динамические массивы*):

```
unit Unit1;
interface
.....
.....
Var
  x : array of real;
  y : array of real;
  t : textfile; // объявили файловую переменную
  Procedure read_text; // объявили переменную процедурного типа
.....
.....
implementation
{$R *.dfm}

Procedure read_text; // реализация переменной процедурного типа
begin
  SetLength(x,1);
  SetLength(y,1);
  i:=0;
  while not eof(t) do begin // читаем файл до конца
    if length(x)<=i then begin // увеличиваем длину массива
```

```

        setlength(x, length(x)+1);
        setlength(y, length(y)+1);
    end;
    readln(t, x[i],y[i]); // читаем файл по строкам
    inc(i);
end; //while
closefile(t);
end; // Procedure read_text
.....
.....
if OpenFileDialog1.ShowDialog = system.Windows.Forms.DialogResult.OK then begin
    assignfile(t,openFileDialog1.filename); //передали в файловую переменную адрес файла
    reset(t); // команда на чтение-запись сначала
    read_text; // обращение к процедуре чтения
end;
.....
.....
end.

```

Типизированные файлы

Типизированные файлы - это файлы, в которых содержатся записи одного типа фиксированной длины. Работа с такими файлами не сложнее, чем работа с текстовыми файлами. Добавляется только одно новое свойство. Если текстовые файлы чисто последовательные, то в типизированных файлах можно перемещаться на любую запись и затем, производить чтение, редактирование данной записи с последующим сохранением в то же место файла. До недавнего времени такие файлы называли как '*файл прямого доступа*'

Типизированный файл объявляется в разделе описания переменных Var - в интерфейсной или исполняемой части программы:

<имя файла> : file of <тип файла>

Например, в программе объявлена следующая *запись* :

```

type
    M_cod = record // тип_имени_записи
        data_set: string[20];
        adr : string[100]
        obgect : string[50];
        Tip_obg : string[30];
        n_obg : string[10];
        n_swid : string[10];
        wlad : string[50];
    end; // type

```

.....

var

cod : M_cod; // <имя записи> : <тип_имени_записи>;

index_file : file of M_cod; //объявлен типизированный файл

Требуется эти данные сохранить в файл. Зададим имя данного файла и его тип. В разделе описания переменных делаем объявление -

index_file : file of M_cod;

где

index_file - имя созданного файла;

M_cod - тип файла, соответствующий типу объявленной записи - **record** ;

В качестве типов не могут использоваться динамические структуры, такие как динамические массивы, длинные строки, указатели. Все записи должны быть одинаковой длины и не указывать на внешние данные. Работа с типизированным файлом аналогична работе с текстовым файлом, т.е. файл открывается теми же функциями: assign, reset, rewrite. Для чтения/записи типизированного файла, в отличии от текстового , используются ***только функции Read и Write !!!***

В отличии от текстовых файлов, функция Reset в типизированных файлах, открывает последний для чтения/записи с любой позиции.

Для работы с типизированными файлами добавлены новые функции:

Seek	Используется для перемещения указателя на нужную запись.
FileSize	Возвращает число записей в файле (но не длину файла !)
FilePos	Возвращает текущую позицию в файле
Truncate	Обрезает файл от текущей позиции

Рассмотрим примеры работы с типизированным файлом:

Пример 1: *Открываем существующий файл и читаем его. Если файл отсутствует, то создаем его.*

.....

.....

Interfase

.....

type

M_cod = record // объявили тип записи

n_zap : longint;

data_set: longint;

n_grup : string[30];

grup : string[10];

Tip_obg : string[30];

```

    obgect : string[30];
    n_obg : byte;
    n_swid : string[10];
    wlad : string[10];
end;
.....
Var
cod : M_cod; // Задали имя записи
index_file : file of M_cod; // задали имя типизированному файлу
fp, FPos : longint; // число записей в файле, номер записи
path : string ;// адрес расположения данных + имена файлов
.....
.....
implementation
.....
.....
Procedure <открытие и чтение типизированного файла>
begin
    fp:=0;
    Assignfile(index_file,path+'.ind');//указан абсолютный адрес
    {$I-} //ключ - проверка на наличие файла
    Reset(index_file);
    {$I+} //ключ - закрыть проверку на наличие файла

    if (IOResult <> 0) then begin // IOResult - системная //переменная - содержит код ошибки. Если
успешно, то IOResult=0.
        Rewrite(index_file); // файл отсутствует на диске. //Создаем новый для записи и чтения
        fp:=-1;
        exit; // покинули процедуру
    end
    else begin
        fp:=filesize(index_file); //определяем количество записей
    end;
    for j:=0 to fp-1 do begin // последовательно читаем записи и //что то с ними делаем
        read(index_file,cod);
        .....
        .....
        здесь что то делаем с записями
        .....
        .....
    end;//for
end;// procedure

```

Пример 2. Поиск конца файла и его дополнение

```

.....
.....
seek(index_file,filesize(index_file));// нашли конец файла
write(index_file,cod);// дополнили файл
.....
.....

```

Пример 3. Поиск нужной записи , ее чтение и перезапись

```
.....  
.....  
seek(index_file,FPos);// позиционируем файл на нужную запись по //ее номеру - FPos  
with cod do read(index_file,cod);//читаем запись  
.....  
здесь что то делаем с записью  
.....  
seek(index_file,FPos); //позиционируем файл на //отредактированную запись по ее номеру - FPos  
write(index_file,cod);//перезаписываем  
.....  
.....
```

Ключевое слово **with** устанавливает значения полей записи с именем COD.

НЕТИПИЗИРОВАННЫЕ (двоичные) ФАЙЛЫ.

Нетипизированные файлы позволяют открывать произвольные файлы в двоичном режиме и осуществлять побайтовый доступ. Данный тип файлов не имеет никакого преимущества перед другими методами работы с файлами.

Такие файлы похожи на типизированные, но отличаются тем, что тип не указывается. Данные читаются целыми частями — по несколько сот или тысяч байт. Это, пожалуй, самый быстрый доступ к файлам. Но разбираться в этих данных, если неизвестна структура данных, придется вручную. Такого автоматического разбора по переменным, как в типизированном файле, здесь нет.

Для нетипизированных файлов нам доступны Rewrite и Reset - равно как и для типизированных файлов. Но тут есть одно важное отличие: для нетипизированных файлов эти процедуры принимают два параметра. Первый параметр, как обычно, файловая переменная, а второй параметр - размер блока. Размер блока измеряется в байтах и является аналогом *размера записи у типизированных файлов*.

Второй параметр, размер блока, является опциональным, его можно не указывать. Если не указывать размер блока, то размер блока по умолчанию будет 128 байт.

Для чтения и записи в нетипизированный файл вместо Read и Write используются функции BlockRead и BlockWrite. Обе они используются одинаково: первый параметр - файловая переменная, второй параметр - что пишем/читаем, третий параметр - сколько пишем/читаем (в блоках).

Следует заметить, что достаточно часто вместо использования типизированного файла оказывается проще открыть файл в двоичном режиме (как нетипизированный) и загрузить/сохранить все данные.

Примеры работы с файлом.

Пример 1: чтение/запись файла с использованием цикла.

type

TData = real; // запись или что угодно - элемент нашей БД.

end;

var

Data: **array of** TData;

F: **file of** TData;

Index: Integer;

begin

// <- заполнение переменной Data

.....

// Запись:

AssignFile(F, 'Test.bin');

Rewrite(F);

for Index := 0 **to** High(Data) **do** Write(F, Data[Index]);

// High - системная функция, возвращает номер последнего элемента массива

// Low - первого элемента

CloseFile(F);

// Чтение:

AssignFile(F, 'Test.bin');

Reset(F);

SetLength(Data, FileSize(F));

for Index := 0 **to** High(Data) **do** Read(F, Data[Index]);

CloseFile(F);

end;

Пример 2: чтение/запись файла с использованием BlockRead и BlockWrite.

var

Data: array of real;

F: file;

begin

// <- заполнение Data

.....

// Запись:

AssignFile(F, 'Test.bin');

Rewrite(F, SizeOf(Data));

BlockWrite(F, Pointer(Data)^, Length(Data));

CloseFile(F);

```
// Чтение:  
AssignFile(F, 'Test.bin');  
Reset(F, SizeOf(TData));  
SetLength(Data, FileSize(F));  
BlockRead(F, Pointer(Data)^, Length(Data));  
CloseFile(F);  
end;
```

Здесь `Pointer(Data)^` - указатель на массив (адрес памяти начала массива).

ДОПОЛНИТЕЛЬНЫЕ ФУНКЦИИ РАБОТЫ С ФАЙЛАМИ

К функциям для работы с файлами, рассмотренными выше, добавились функции, работающие с дескриптором файла. Они рассматривают файл как двоичный.

Файловые функции, описанные в модуле SYSUTILS, приведены в таблице:

FileOpen(const FileName: string; Mode: Word): Integer;	Открывает существующий FileName файл в режиме Mode (см. примеч. 1). Значение, возвращаемое в случае успеха, — дескриптор открытого файла(целое положительное число). В противном случае — код ошибки DOS.
FileCreate(const FileName: string): Integer;	Создает файл с именем FileName. Возвращает то же, что и FileOpen.
FileRead(Handle: Integer; var Buffer; Count: Longint): Longint;	Считывает из файла с дескриптором Handle Count байт в буфер Buffer. Возвращает число реально прочитанных байт или (-1) при ошибке.
FileWrite(Handle: Integer; const Buffer; Count: Longint): Longint;	Записывает в файл с дескриптором Handle Count байт из буфера Buffer. Возвращает число реально записанных байт или -1 при ошибке.
FileSeek(Handle: Integer; Offset: Longint; Origin: Integer): Longint;	Позиционирует файл с дескриптором Handle в новое положение. При Origin = 1,2,3 положение смещается на Offset байт от начала файла, текущей позиции и конца файла соответственно. Возвращает новое положение или -1 при ошибке.
FileClose(Handle: Integer);	Закрывает файл с дескриптором Handle.
FileAge(const FileName: string): Longint	Возвращает значения даты и времени создания файла или -1, если файл не существует.
FileExists(const FileName: string): Boolean;	Возвращает True если файл FileName существует и найден.
FindFirst(const Path: string; Attr: Integer; var SearchRec: TSearchRec):	Ищет первый файл, удовлетворяющий маске поиска,

Integer;	заданной в Path и с атрибутами Attr (см. примеч. 2). В случае успеха заполняет запись SearchRec (см. примеч. 3) и возвращает 0, иначе возвращает код ошибки DOS
----------	---

FindNext(var SearchRec: TSearchRec): Integer;	Продолжает процесс поиска файлов, удовлетворяющих маске поиска. Параметр SearchRec должен быть заполнен при помощи FindFirst. Возвращает 0, если очередной файл найден, или код ошибки DOS. Изменяет SearchRec.
FindClose (var SearchRec: TSearchRec);	Завершает процесс поиска файлов, удовлетворяющих маске поиска.
FileGetDate(Handle: Integer): Longint;	Возвращает время создания файла с дескриптором Handle (в формате DOS) или -1, если дескриптор недействителен.
FileSetDate(Handle: Integer; Age : Longint) ;	Устанавливает время создания файла с дескриптором Handle (в формате DOS).
FileGetAttr(const FileName: string): Integer;	Возвращает атрибуты (см. примеч. 2) файла с именем FileName или код ошибки DOS, если файл не найден.
FileSetAttr(const FileName: string; Attr: Integer) : Integer;	Устанавливает атрибуты файла с именем FileName.
DeleteFile(const FileName: string): Boolean;	Уничтожает файл с именем FileName и в случае успеха возвращает True
RenameFile(const OldName, NewName: string): Boolean;	Переименовывает файл с именем OldName в NewName и возвращает True в случае успеха.
ChangeFileExt(const FileName, Extension: string): string;	Изменяет расширение в имени файла FileName на Extension и возвращает новое значение FileName. Имя файла не изменяется.
ExtractFileLePath(const FileName: string): string;	Извлекает из строки с полным именем файла FileName часть, содержащую путь к нему.

ExtractFileName(const FileName: string): string;	Извлекает из строки с полным именем файла FileName часть, содержащую его имя и расширение.
ExtractFileExt(const FileName: string): string;	Извлекает из строки с полным именем файла FileName часть, содержащую его расширение.
ExpandFileName(const FileName: string):	Возвращает полное имя файла FileName, добавляя при необходимости путь к нему
FileSearch(const Name, DirList: string): string;	Производит поиск файла с именем Name в группе каталогов, заданных параметром DirList. Имена каталогов должны отделяться друг от друга точкой с запятой. Возвращает в случае успеха полное имя файла или пустую строку, если файл не найден.

DiskFree(Drive: Byte) : Longint;	Возвращает количество в байтах свободного места на заданном диске. Значение параметра Drive: 0 — для текущего диска, 1 — для A, 2 — для B и т. д. Если параметр неверен, функция возвращает -1.
DiskSize(Drive: Byte) : Longint;	Возвращает размер диска Drive в байтах. Параметр Drive означает то же, что и в DiskFree.
FileDateToDateTime(FileDate: Longint): TDateTime;	Преобразует дату и время в формате DOS в принятый в Delphi формат TDateTime.
DateTimeToFileDate(DateTime: TDateTime): Longint;	Преобразует дату и время из формата TDateTime в формат DOS.

Примечания:

1. Допустимы следующие режимы открытия файлов:

Режим	Значение	Что означает
fmOpenRead	\$0000	Открыть только для чтения.

fmOpenWrite	\$0001	Открыть только для записи.
fmOpenReadWrite	\$0002	Открыть для чтения и записи.
fmShareCompat	\$0000	Обеспечить совместимость со старой моделью доступа к файлам.
fmShareExclusive	\$0010	Запретить другим доступ к файлу.
fmShareDenyWrite	\$0020	Запретить другим запись в файл.
fmShareDenyRead	\$0030	Запретить другим чтение файла.
fmShareDenyNone	\$0040	Разрешить другим все виды доступа.

2. Файлы могут иметь следующие атрибуты:

faReadOnly	\$01
faHidden	\$02
faSysFile	\$04
faVolumeID	\$08
faDirectory	\$10
faArchive	\$20
faAnyFile	\$3F

3. Формат структуры **TSearchRec**:

```

TSearchRec = record
    Fill: array[1..21] of Byte;
    Attr: Byte;
    Time: Longint;
    Size: Longint;
    Name: string [12] ;
end;
```

Приведем типовой вариант организации групповой обработки файлов при помощи функций FindFirst, FindNext и FindClose. В нем в список добавляются имя и длина всех файлов в каталоге с именем C:\DELPHI:

var

SearchRec: TSearchRec;

I : Integer;

List : TStringList;

begin

List := TStringList.Create;

I := FindFirst('c:*.*.*', faAnyFile, SearchRec);

while I = 0 **do begin**

List.Add(Format (' File %s has length %d bytes ',
SearchRec.Name, SearchRec.Size));

I := FindNext(SearchRec);

end;

FindClose(SearchRec);

List.Free;

end;

При поиске файла наиболее вероятным является код ошибки -18 (файл не найден).

Приложения.

Задание 1.

Вычислить расстояния до горизонта моря, с высоты роста человека, стоящего на берегу у кромки воды.

Цель Задания:

1. Знакомство с работой текстового редактора TextBox для ввода исходных данных в программу.
2. Научиться преобразовывать текстовые данные к заданному типу данных.
3. Работа с компонентами Button.
4. Работа с компонентом ListBox, для визуализации результатов вычислений..
5. Научиться составлять простейшие операторы для решения задачи.

Для решения данной задачи используем формулу из геодезии “Вычисление поправки в превышение за кривизну земли”

$$dh = \frac{S^2}{2R}$$

Преобразуем данную формулу относительно S :

$$S = \sqrt{dh \times 2 \times R}$$

Где:

dh – рост человека в метрах;

R - радиус земли = 6371км = 6371000 метра;

S - вычисляемое расстояние до горизонта - метры.

Программирование задач под Windows состоит из двух этапов.

1. ***Разработка интерфейсной части программы (рабочего стола);***
2. ***Разработка программного кода.***

Разработка интерфейсной части программы.

Первое, что необходимо сделать, установить в инспекторе объектов свойство формы **Form1 – WindowState – в Maximized**, как на рисунке 18. При запуске программы, данное свойство задает размер формы, равный размеру экрана монитора.

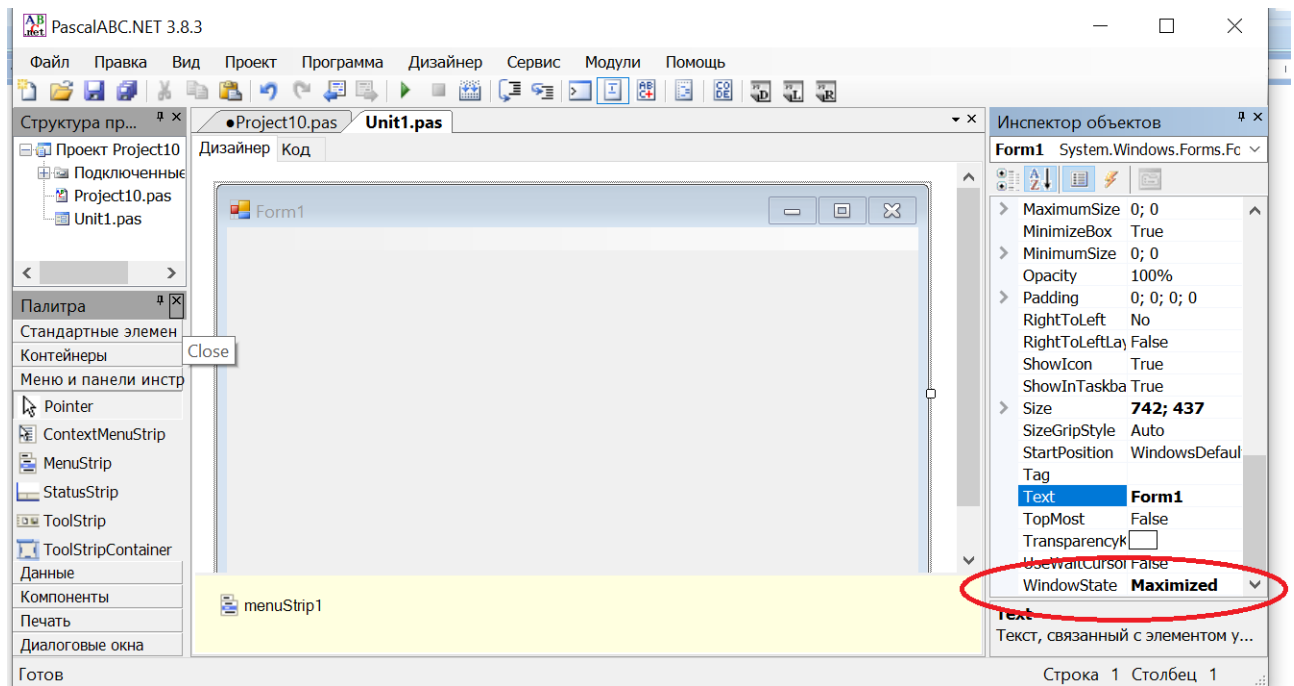


Рис.18

Второе – создать строчное меню для управления интерфейсом.

Для этого входим в меню «Палитра» и выбираем раздел «Меню и панели инструментов». В открывшемся разделе щелкаем мышкой по строке «MenuStrip» - строчное меню, а затем щелкаем мышкой по форме.

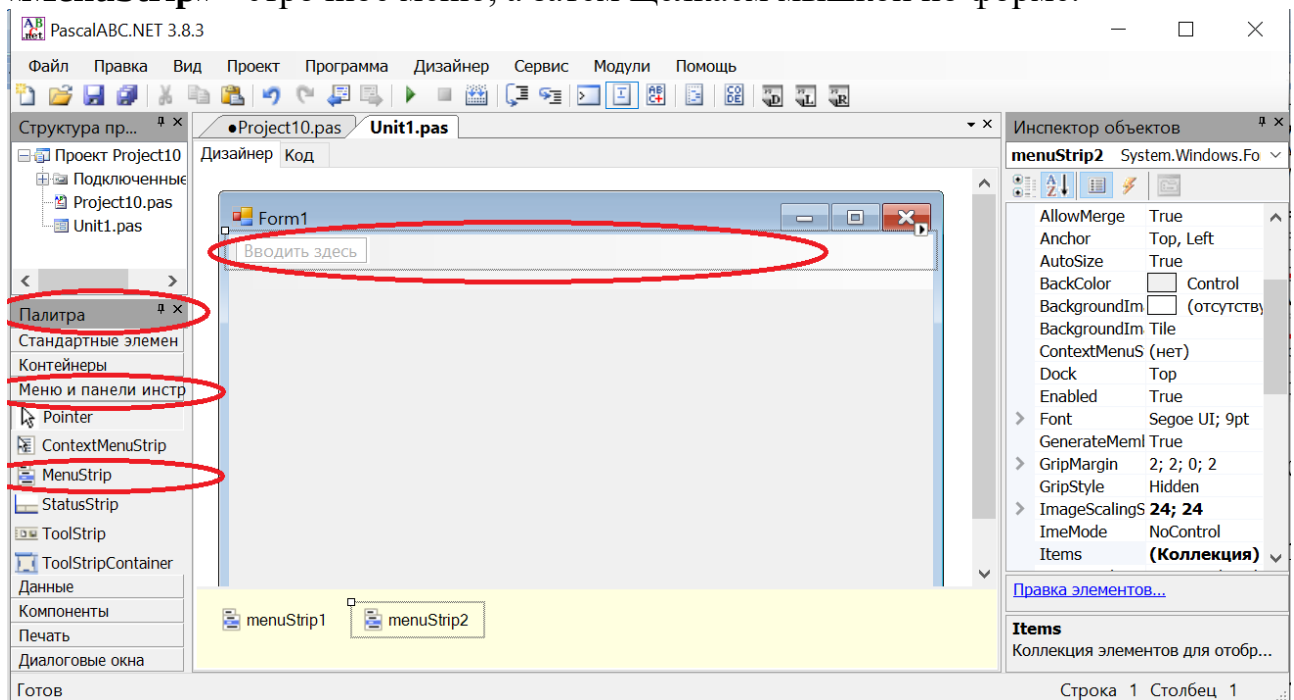


Рис.19

На форме отобразится полоса строчного меню, готовая к разработке – рис.19.

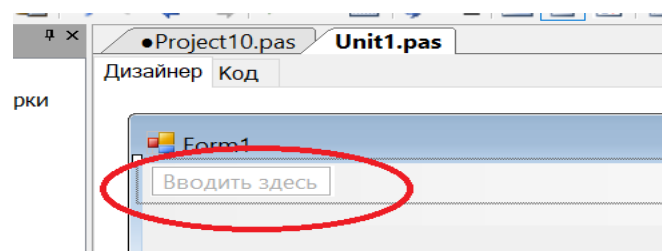


Рис.20

Щелкнем мышкой по окошку «Введите здесь» - рис.20 – и зададим первый пункт строчного меню, написав с клавиатуры – ‘Задачи’. После чего, в выпавшем меню определим первую задачу – ‘Горизонт’ – рис.21.

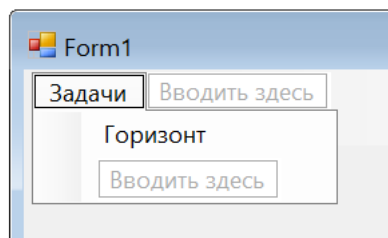


Рис.21

Затем создаем кнопку «Выход» (Рис.22), для корректного завершения работы программы, щелкнув по кнопке справа от задач.

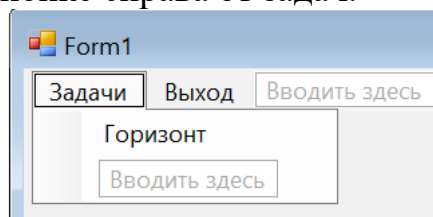


Рис.22

Запрограммируем кнопку «Выход». Для этого необходимо мышкой дважды щелкнуть по данной кнопке. В результате, в разделе **implementation**, автоматически сгенерируется процедура обработки события Click – щелчка мышки по компоненту.

Unit Unit1;

interface

uses System, System.Drawing, System.Windows.Forms;

type

```

    Form1 = class(Form)
        procedure toolStripMenuItem2_Click(sender: Object; e:
EventArgs);
    {$region FormDesigner}
    internal
        {$resource Unit1.Form1.resources}
        menuStrip1: MenuStrip;
        toolStripMenuItem1: ToolStripMenuItem;
        toolStripMenuItem3: ToolStripMenuItem;
        toolStripMenuItem2: ToolStripMenuItem;
        {$include Unit1.Form1.inc}
    {$endregion FormDesigner}
    public
        constructor;
        begin
            InitializeComponent;
        end;
    end;
end;
```

implementation

```
procedure Form1.toolStripMenuItem2_Click(sender: Object;  
e: EventArgs);  
begin  
  
end;  
  
end.
```

Рассмотрим приведенный листинг подробно.

В разделе написания кода – **implementation** – сгенерировалась процедура обработки щелчка мышки по компоненту «Выход», в которую запишем код завершения работы программы:

```
procedure Form1.toolStripMenuItem2_Click(sender:  
Object; e: EventArgs);  
begin  
    Halt; // оператор - завершающий работу программы.  
end;
```

В разделе описания типов – **type** – автоматически прописалась строка

```
procedure toolStripMenuItem2_Click(sender: Object; e:  
EventArgs);
```

В разделе описания компонентов - **internal** – автоматически прописываются все компоненты, находящиеся на форме,

```
internal  
{$resource Unit1.Form1.resources}  
menuStrip1: MenuStrip;  
toolStripMenuItem1: ToolStripMenuItem;  
toolStripMenuItem3: ToolStripMenuItem;  
toolStripMenuItem2: ToolStripMenuItem;
```

В данном случае – рис.22 – на форме присутствуют 4 компонента:

menuStrip1 – разработчик строчного меню

и 3 кнопки меню:

toolStripMenuItem1 – «Задачи»

toolStripMenuItem2 – «Выход»

toolStripMenuItem3 – «Горизонт»

Кроме этого, в инспекторе объектов (рис.23), для события нажатия на кнопку меню **toolStripMenuItem2**, пометилось событие Click.

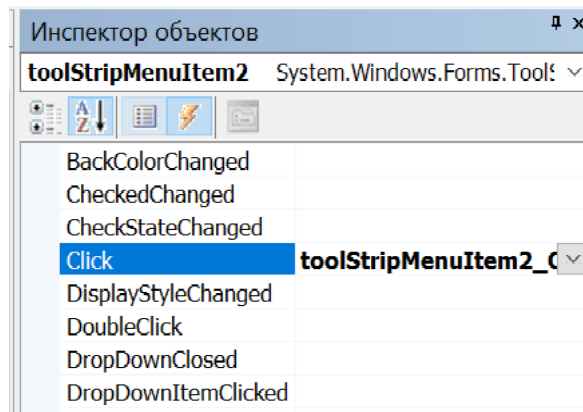


Рис.23

Данную задачу выполним в отдельном окне (рис.24), которое будет представлять собой контейнер, в виде панели, содержащий все необходимые компоненты для ее решения :

TextBox – для ввода информации – ‘Введите рост в метрах’

ListBox – для отображения результата решения задачи

Два кнопки – Button – ‘Выполнить’ и ‘Скрыть панель’

И компонента Label – ‘Горизонт’

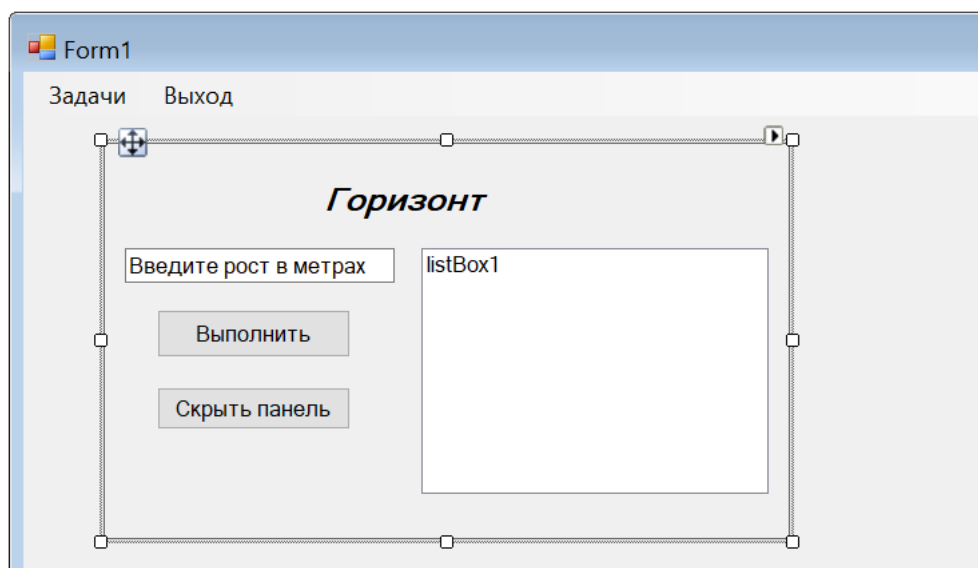
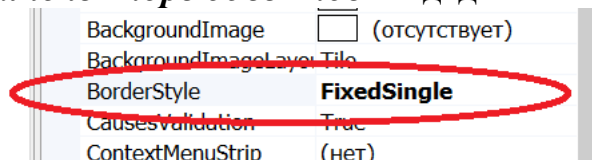


Рис.24

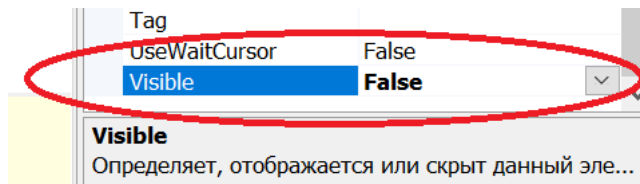
Все необходимые компоненты находим в окне «Палитра».

Последовательность действий:

1. Щелкаем по кнопке «Контейнеры». В открывшемся окне находим компонент «Panel» и щелкаем по нему мышкой. Затем щелкаем мышкой по форме – Form1. На форме отобразилась Панель в виде квадрата. В «*инспекторе объектов*» зададим 2 свойства для панели



Данное свойство отвечает за тип окантовки панели. И



Данное свойство отвечает за видимость компонента – если Visible = False, то компонент становится не видимым, если true, то видимым.

3. Далее выбираем в окне «Палитра» кнопку «Стандартные элементы управления».

В выпавшем окне находим компонент Label и аналогичным образом, перетаскиваем его на панель, Данный компонент будет заголовком панели (рис.24). Для этого в «*инспекторе объектов*» в свойстве Text запишем имя решаемой задачи – «Горизонт»



Для выбора типа и размера шрифта воспользуемся свойством Font «*инспектора объектов*».

Далее в «Стандартных элементах управления» находим компонент TextBox и перетаскиваем его на нашу панель. В свойстве Text «*инспектора объектов*» запишем подсказку, что нужно ввести для решения задачи – «Введите рост в метрах» (рис.24).

Затем выбираем компонент Button и перетаскиваем 2 кнопки на панель, В свойстве Text «*инспектора объектов*» запишем названия кнопок – «Выполнить» (задачу) и «Скрыть панель» (рис.24).

Далее перетаскиваем на панель компонент ListBox, для отображения результатов решения задачи и сообщений.

Рабочий стол подготовлен. Приступим к написанию кода.

Объявим глобальные переменные, необходимые для решения задач проекта. Глобальные переменные объявляются в конце раздела **interface**.

```
var ss,ss1,ss2:string;
    sr,wis: integer;
    Procedure centr(p : panel);
```

implementation

Здесь объявлены 3 переменных текстового типа **ss,ss1,ss2:string**; Данные переменные необходимы для преобразования результатов вычислений в текстовый вид.

Две переменных целочисленного типа - **sr,wis: integer**;

Sr – соответствует ширине формы.

Wis - соответствует высоте формы..

Координаты левого верхнего угла компонента отвечают за расположение его на форме.

И одна переменная процедурного типа

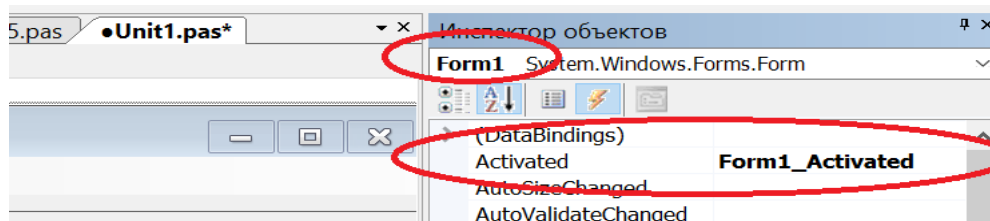
- **Procedure centr(p : panel) ;**

С помощью данной процедуры будем выполнять центрирование панели, в которой будет выполняться задача, относительно экрана монитора.

Разработка кода выполняется в разделе **implementation**.

В процессе выполнения задания, ширина и высота формы будет оставаться постоянными. Поэтому найдем данные величины один раз и поместим их в глобальные переменные – **Sr и Wis**.

Для этого, в инспекторе объектов, для **Form1**, найдем событие **Activated** и щелкнем по нему мышкой дважды.



В разделе **Implementation** отобразится обработчик события :

```
procedure Form1.Form1_Activated(sender: Object; e:
EventArgs) ;
begin

end;
```

При активации формы (запуске приложения), данная процедура выполняется автоматически.

Найдем ширину и высоту формы.

```
procedure Form1.Form1_Activated(sender: Object; e:
EventArgs) ;
begin
    sr:=self.size.width; //ширина формы
    wis:=self.size.Height; //высота формы
end;
```

В данной проседуре:

Self – активируемый компонент - в нашем случае Form1.

size.width – размер по ширине.

size.Height – размер по высоте.

Теперь напомним код процедуры **centr**, в которой найдем координаты левого верхнего угла панели.

Координаты левого верхнего угла компонента отвечают за расположение его на форме.

Данную процедуру пишем самостоятельно, т.е. руками.

```

Procedure centr(p : panel);
begin
  p.Left:=(sr div 2)-(p.Width div 2);
  p.Top:=(wis div 2)-(p.Height div 2);
end;

```

В данной проседуре:

P – панель, которую получила процедура в качестве фактического параметра;

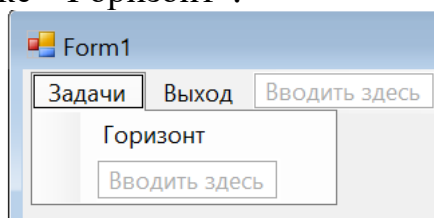
sr div 2 – координата центра формы по оси **X**;

wis div 2 - координата центра формы по оси **Y**;

p.Width div 2 - координата центра панели по оси **X**;

p.Height div 2 - координата центра панели по оси **Y**;

Далее отобразим панель на форме. Для этого, в строчном меню дважды щелкнем мышкой по кнопке “Горизонт”.



В разделе **implementation** автоматически сгенерируется обработчик события щелчка мыши по кнопке “Горизонт”.

```

procedure Form1.toolStripMenuItem4_Click(sender: Object;
e: EventArgs);
begin

end;

```

В данный обработчик запишем следующий код:

```

procedure Form1.toolStripMenuItem4_Click(sender: Object;
e: EventArgs);
begin
  centr(Panell1);
  Panell1.Visible:=true;
end;

```

В данной проседуре:

centr(Panell1) - в процедуру **centr** передали соответствующую панель **Panell1**;

Panell1.Visible:=true – панель сделали видимой.

Далее, активируем буттон «Выполнить». Для этого дважды щелкнем мышкой по бутону. Сгенерируется обработчик событий:

```

procedure Form1.button1_Click(sender: Object; e:
EventArgs) ;//горизонт
begin

end;

```

В который запишем алгоритм вычисления расстояния - $S = \sqrt{dh \times 2 \times R}$
 Для этого, и процедуре объявим необходимые *локальные переменные*, необходимые для решения задачи:

h – рост наблюдателя;
 S – расстояние до горизонта;
 R – радиус земли.

```

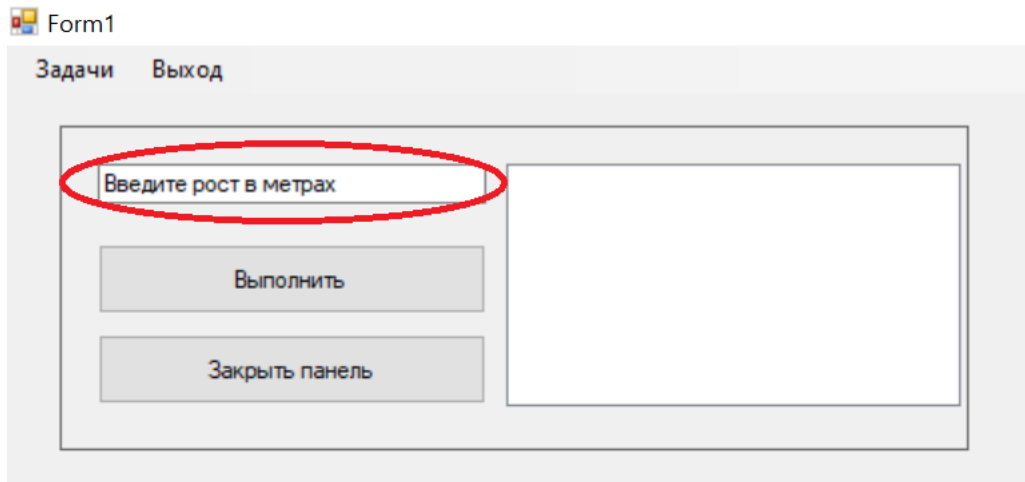
procedure Form1.button1_Click(sender: Object; e:
EventArgs) ;//горизонт
var H,s,r: real;
    n : integer;
begin
    r:=6371000;
    val (Textbox1.Text,h,n) ;//
    if n<>0 then begin
        listbox1.Items.Add('Ошибка преобразования роста');
        exit;
    end;
    s:=sqrt(h*2*r) ;
    str(s:10:1,ss) ;
    listbox1.Items.Add('S = '+ss+' м');
end;

```

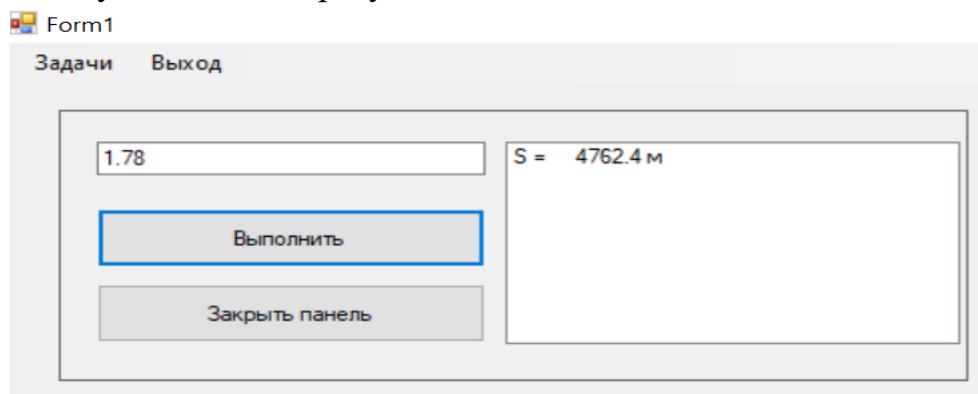
Здесь

listbox1 – компонент типа «Блокнот» в Windows;
 Items – строка listbox1;
 Add – добавить.
 ('Ошибка преобразования роста ') – текст записываемый
 в listbox1;

При запуске программы, в **TextBox1** – обведен красным – запишем свой рост в метрах – например 1.78 – один метр и 78 сантиметров.



После чего, щелкнем мышкой по кнопке «Выполнить».
В **listbox1** будет выведен результат вычислений.



Для того, чтобы скрыть панель, дважды щелкнем мышкой по кнопке «Заккрыть панель» и свойству **Visible** присвоим значение False;

```
procedure Form1.button2_Click(sender: Object; e: EventArgs) ;  
begin  
    Panel1.Visible:=false;  
end;
```

Разработка программы завершена.

Задание 2.

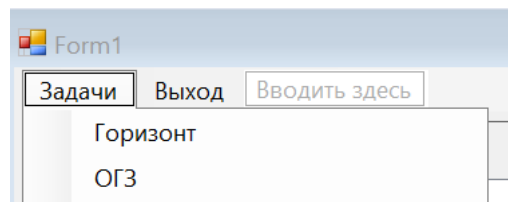
Обратная геодезическая задача.

Все дальнейшие задачи делаем в том же проекте.

Цель Задания:

1. Закрепить навыки преобразования типов данных.
2. Знакомство с логическими операциями.
3. Преобразование радианной меры угла в градусную, для визуализации результатов решения задачи.

В строчное меню добавим новую кнопку



и подпишем ее ОГЗ – обратная геодезическая задача.

На форму бросим новую панель – Panel2. Свойство Visible для панели установим в False. На панель бросим 4 TextBox_а и 2 буттона, как и предыдущем задании. Для визуализации результата – ListBox.

