**Записи и вариант. Сетка строк TStringGrid**

**Запись - пользовательский тип данных**

В Паскале предусмотрены не только стандартные типы, можно использовать и новые типы данных, определенные пользователем. Один из таких типов - запись.

**Запись** *(анг. record) - это структура данных, определенная пользователем из различных стандартно-типовых полей.*

По сути, запись является мини-объектом, который имеет свойства (поля), но не имеет методов и событий. Еще запись можно сравнить с массивом - в ней также хранятся различные данные, объединенные в одной переменной, только в отличие от массива, эти данные могут быть разнотипными.

Записи часто применяют для сохранения и загрузки различных данных, например, настроек программы. С помощью записей можно даже сделать небольшую базу данных, используя типовой файл, что мы и проделаем в следующей лабораторной работе.

Использование записей происходит в два этапа: вначале объявляется сама запись, затем объявляется переменная, которой присваивают тип этой записи. Кстати, тип записи можно присвоить не только переменной, но и массиву, и типизированному файлу.

Записи объявляются в разделе type перед разделом var, и могут иметь как локальную область видимости (запись объявлена внутри подпрограммы), так и глобальную. Синтаксис объявления следующий:

type

<имя\_записи> = record

<поле1>: <тип\_поля>;

<поле2>: <тип\_поля>;

…;

<полеN>: <тип\_поля>;

end;

Здесь, имя\_записи - имя, которым пользователь называет новый созданный тип данных. Поле - имя поля записи, фактически, это переменная внутри другой переменной. Тип\_поля - один из стандартных типов. Как уже говорилось, объявлять тип string в записи не запрещается, однако использовать такую запись в типизированном файле не получится. Обусловлено это тем, что при создании типизированного файла, компилятор должен точно знать, сколько байт на каждую запись он должен отвести в файле. Строка string является динамическим символьным массивом, то есть, заранее узнать её длину невозможно. Однако, можно решить и эту проблему, если поставить ограничитель на строку, например, string[50] - строка из 50 символов. В этом случае, типизированный файл можно создать. И если строка будет пустая, или в ней до 50 не будет хватать символов, то часть этой записи в файле просто окажется пустой, хотя место под нее будет выделено. Правда, использование кириллицы в кодировке UTF-8 создает некоторые трудности, но и они решаемы.

После того, как запись объявлена, нужно создать переменную, присвоив ей тип этой записи. Делается это обычным образом, в локальном или глобальном разделе var:

var

NewPerem: <имя\_записи>;

После чего, можно обращаться к отдельному полю этой записи, разделяя его от имени переменной точкой, например:

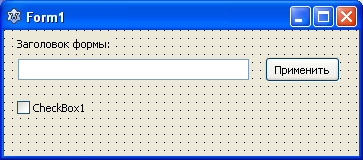
NewPerem.Pole1:= 10;

Давайте освоим работу с записями на примере программы, которая сохраняет свои настройки, такие как положение и размеры формы, состояние некоторых компонентов, в типизированном файле, а при последующей загрузке программа считывает эти настройки и подстраивается под них.

Откройте **Lazarus** с новым проектом. Сразу договоримся, что поскольку проект очень простой и пробный, то имена формы, модуля, проекта и компонентов мы оставим по умолчанию. Сразу же сохраните проект в папку **24-01**. Теперь, на форме нам потребуются:

* Метка TLabel;
* Строка TEdit;
* Кнопка TButton;
* Флажок TCheckBox.

В Caption метки напишите "Заголовок формы:". Очистите Text у Edit1. В Caption кнопки напишите "Применить". А с флажком можно ничего не делать. В результате у вас должна получиться примерно такая форма:



Форма проекта

Теперь самое главное - нам нужно описать запись, в которую затем сохраним параметры формы. Для этого нужно создать раздел type ПЕРЕД глобальным разделом var, тем самым, в котором описана переменная формы:

var

Form1: TForm1;

И не смущайтесь, что выше раздел type уже был, таких разделов может быть много. Код нашей вставки следующий:

type

myForm = record //запись

Left: integer; //левая граница формы

Top: integer; //верхняя

Height: integer; //высота формы

Width: integer; //ширина

Caption: string[100]; //заголовок

Checked: boolean; //состояние флажка CheckBox1

wsMax: boolean; //окно развернуто?

end; //record

Как видите, внутри раздела type мы описали новый, пользовательский (наш) тип данных - запись. Теперь нам нужна переменная этого типа, с которой мы будем работать. Нашей программе нужно сохранять свои настройки при выходе, делать это лучше в событии OnClose формы. Но ей также нужно и считывать эти настройки при загрузке, делать это будем в событии OnCreate формы. Поскольку переменная нам будет нужна и там, и там, то значит, опишем её в глобальном разделе var, сразу ПОСЛЕ описания переменной Form1:

var

Form1: TForm1;

MyF: myForm; //наша переменная-запись

Теперь займемся кодированием. Прежде всего, нам нужно "научить" программу менять свой заголовок. Для этого сгенерируйте событие OnClick для кнопки "**Применить**", его код:

procedure TForm1.Button1Click(Sender: TObject);

begin

if Edit1.Text <> '' then Form1.Caption:= Edit1.Text;

end;

Тут всё просто: мы проверяем - не пуста ли строка Edit1, и если не пуста, то присваиваем её текст свойству Caption формы, тем самым, меняя заголовок окна. Пойдем далее. А далее нам нужно научить программу сохранять свои настройки. Для этого выделите форму, щелкнув по её любому свободному месту. Теперь перейдите на вкладку **События Инспектора объектов**, и двойным щелчком сгенерируйте событие формы OnClose, его код:

procedure TForm1.FormClose(Sender: TObject; var CloseAction: TCloseAction);

var

f: file of myForm; //типизированный файл для сохранения данных

begin

//сначала настроим переменную-запись:

//не сохраняем размеры и положение, если окно wsMaximized:

if not (Form1.WindowState = wsMaximized) then begin

MyF.Left:= Form1.Left;

MyF.Top:= Form1.Top;

MyF.Height:= Form1.Height;

MyF.Width:= Form1.Width;

MyF.wsMax:= False;

end

else MyF.wsMax:= True;

//остальные настройки:

MyF.Caption:= Form1.Caption;

MyF.Checked:= CheckBox1.Checked;

//теперь создаем или перезаписываем файл настроек:

try

AssignFile(f, 'MyProg.dat');

Rewrite(f);

Write(f, MyF); //записываем в файл данные из записи

finally

CloseFile(f);

end;

end;

Давайте разберем этот код. В разделе var мы описали переменную f, которая будет ассоциироваться с типизированным файлом, имеющим тип нашей записи. Видите, мы присвоили наш тип и переменной, и типизированному файлу!

Далее нам требуется сохранить в переменную-запись настройки нашей формы, но тут есть одно НО: если окно программы было развернуто, то такие данные, как левая и верхняя граница окна, его высота и ширина становятся недействительными. Поэтому мы пошли на хитрость: если окно не было развернуто, то мы сохраняем эти параметры и, кроме того, полю wsMax присваиваем False, тем самым указывая, что окно не было развернуто. Если же окно было развернуто, то мы присваиваем этому полю True, не трогая границы окна:

if not (Form1.WindowState = wsMaximized) then begin

MyF.Left:= Form1.Left;

MyF.Top:= Form1.Top;

MyF.Height:= Form1.Height;

MyF.Width:= Form1.Width;

MyF.wsMax:= False;

end

else MyF.wsMax:= True;

Как видно из кода, мы обращаемся к отдельным полям записи через точку:

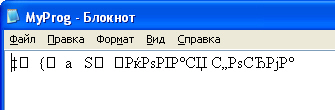
MyF.Left

Остальные параметры (заголовок окна и состояние флажка) нам нужно сохранять в любом случае:

MyF.Caption:= Form1.Caption;

MyF.Checked:= CheckBox1.Checked;

Далее мы работаем с файлом, создавая или перезаписывая его. В файл мы записываем одну-единственную запись - нашу переменную MyF, в которой уже хранятся все нужные настройки, после чего файл закрываем. Забегая вперед, скажу, что если вы попытаетесь открыть такой файл редактором текстов, то увидите в нем двоичный код:



Содержимое файла с настройками

Неискушенный пользователь не рискнет пытаться править такой файл, а если вы случайно чего-то в нем измените, и программа перестанет загружаться, то просто удалите файл - без него программа загрузится с настройками по умолчанию, а при следующем закрытии файл **MyProg.dat** будет создан снова.

Теперь нужно научить программу считывать настройки из файла, и применять их. Сгенерируйте событие формы OnCreate, его код:

procedure TForm1.FormCreate(Sender: TObject);

var

f: file of myForm; //типизированный файл для сохранения данных

begin

//если файла еще нет, просто выходим:

if not FileExists('MyProg.dat') then exit;

//иначе открываем файл и считываем из него настройки в запись:

try

AssignFile(f, 'MyProg.dat');

Reset(f);

Read(f, MyF); //считали данные в переменную типа запись

finally

CloseFile(f);

end;

//теперь настройки нужно применить к форме:

//если окно было развернуто, разворачиваем его, иначе настраиваем края:

if MyF.wsMax then Form1.WindowState:= wsMaximized

else begin

Form1.Left:= MyF.Left;

Form1.Top:= MyF.Top;

Form1.Height:= MyF.Height;

Form1.Width:= MyF.Width;

end;

//остальные настройки:

Form1.Caption:= MyF.Caption;

CheckBox1.Checked:= MyF.Checked;

end;

Вначале мы проверяем - есть ли файл. Если нет (программа открывается впервые), то мы выходим, ничего не считывая и не меняя настроек. В этом случае, все настройки будут, как при разработке формы.

Комментировать работу с файлом я не буду - всё должно быть предельно ясно по прошлой лекции и по предыдущему коду сохранения файла. В нашу переменную-запись MyF считались настройки, сохраненные в файле, и нам осталось только применить их к форме. Делается это как при сохранении, но в обратном направлении - не в запись из формы, а в форму из записи. Также, если окно было развернуто, разворачиваем его, иначе применяем настройки положения и размеров формы.

Теперь сохраните проект, запустите его на исполнение и попробуйте менять положение и размеры окна, разворачивать его - все эти настройки при выходе должны сохраняться, а при следующем запуске - восстанавливаться.

**Вариант**

В Lazarus существует еще один интересный тип данных - variant. Вариантный тип был разработан в Object Pascal для поддержки работы с механизмом OLE (англ. Object Link and Embedding - внедрение и связь объектов). Даже если вы впервые слышите про такой механизм, вы с ним наверняка знакомы, и не раз им пользовались. Каждый раз, когда вы копируете таблицу MS Excel в MS Word, или картинку из Paint в тот же MS Word, или текстовое поле из Word в Excel; другими словами, любой объект, созданный в одной программе, копируете в другую программу - вы используете OLE.

В контексте Lazarus variant может содержать любое число - целое, или вещественное, строку, логическое значение, время-дату, массив, тот же OLE-объект. При этом компилятор сам определяет реальный тип варианта на этапе выполнения программы. Причем сразу же выделяет под переменную этого типа память - если значение не указано, этой переменной присваивается ноль.

Давайте рассмотрим следующий пример:

var

MyV: variant;

begin

MyV:=10; //вариант содержит целое число

MyV:= 2.5; //вариант содержит уже вещественное число

MyV:= 'Какой то текст'; //вариант содержит строку

MyV:= True; //а теперь вариант содержит логическое значение

Как видно из примера, мы присваиваем одной и той же вариантной переменной значения разных типов. При этом компилятор сам определяет наиболее подходящий по значению тип, и переделывает вариантную переменную под него. Например, в строке

MyV:=10;

компилятор найдет наиболее подходящий под значение тип - Byte, и изменит переменную MyV под него. Таким образом, программист может не беспокоится о типах - за него эту работу выполнит компилятор. В вышеописанном примере переменная MyV поменяла свой тип 4 раза!

Однако за всё приходится платить. Переменная типа variant занимает гораздо больше памяти, чем переменная любого другого типа (за исключением строковой переменной, в которую можно поместить оба тома "Война и мир"). Поэтому, используйте тип variant только тогда, когда это действительно необходимо, например, когда на этапе программирования вы не можете заранее определить, данные какого типа придется сохранять в переменную. В других случаях использование типа variant неоправданно.

С переменными типа variant можно производить различные действия, например, складывать. При этом различные типы данных будут преобразовываться под нужный тип, если это возможно. Давайте попробуем работу с переменной типа variant на практике. Откройте **Lazarus** с новым проектом. Не меняя имен по умолчанию, сразу сохраните проект в папку **24-02**. На форму установите только одну кнопку и сгенерируйте для нее событие OnClick. Его код следующий:

procedure TForm1.Button1Click(Sender: TObject);

var

v1, v2, v3: variant;

begin

v1:= '5'; //присвоили символ

v2:= '10'; //теперь строку

v3:= 20; //теперь целое число

v1:= v1 + v2 + v3; //теперь все сложили

ShowMessage(IntToStr(v1));

end;

Не торопитесь запускать эту программу, сначала попробуйте ответить: какое число в итоге выведет сообщение ShowMessage? На первый взгляд, результат будет 35 (5 + 10 + 20). Однако не спешите с выводами, подумайте. Мы к символу '5' прибавили строку '10'. Поскольку все однотипные операции производятся слева-направо, то в результате мы получим строку '510'. А затем мы к этой строке прибавляем целое число 20. Вариантная переменная, которая сначала имела символьный тип, затем строковый, получила целочисленный тип! А если к числу 510 прибавить 20, то сколько получится? Конечно, 530! Сохраните проект, запустите его на выполнение и убедитесь в этом сами.

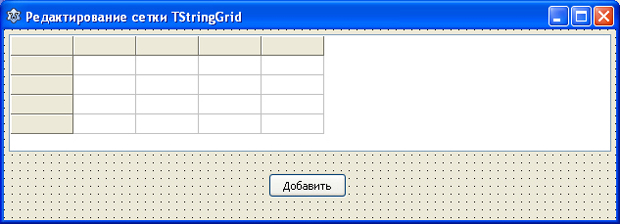
**Сетка строк TStringGrid**

Напоследок познакомимся с еще одним полезным инструментом - сеткой строк TStringGrid. Это сетка, похожая на сетку в электронной таблице MS Excel, она позволяет вводить и редактировать табличную информацию в виде строк и колонок.

Загрузите **Lazarus** с новым проектом. Форму переименуйте в **fMain**, проект сохраните в папку **24-03** под именем **Setka**, а модулю формы дайте имя Main. Чтобы ваша форма соответствовала моей, установите ширину width = 625, а высоту height = 195. В Caption напишите "Редактирование сетки TStringGrid".

Теперь перейдите на вкладку **Additional**, найдите и установите на форму компонент TStringGrid. Рядом с ним вы можете увидеть еще одну сетку, TDrawGrid. Разница между ними в том, что сетка TStringGrid самостоятельно прорисовывает данные в ячейках, а в TDrawGrid это придется делать вручную. Так как TStringGrid удобней, её и будем изучать. У сетки установите ширину 615, а высоту 120 пикселей.

Ниже установите одну простую кнопку **TButton**, в Caption которой напишите "Добавить". В результате, у вас должна получиться вот такая форма:



Форма проекта

Прежде, чем приступить к кодированию, выделите на форме сетку и посмотрите в **Инспекторе объектов** на её свойства. Познакомимся с наиболее значимыми из них.

|  |  |  |
| --- | --- | --- |
| BorderStyle | - стиль обрамления. Может иметь только два значения - с обрамлением и без него. | |
| ColCount | - количество колонок в сетке. По умолчанию их 5. При необходимости, это значение можно изменять программно, увеличивая или уменьшая количество колонок. | |
| DefaultColWidth | - ширина колонок по умолчанию. К сожалению, всем колонкам устанавливается одинаковая ширина, хотя, как правило, разные колонки должны иметь разную ширину. Во время работы программы ширину каждой колонки можно будет изменить программно. | |
| DefaultDrawing | - прорисовка данных по умолчанию. Если стоит True, то компонент сам будет отображать введенные данные, иначе это придется делать программисту. | |
| DefaultRowHeight | - высота строк по умолчанию. Установлено 20 пикселей, но этот размер немного великоват, рекомендую изменить его на 18. Хотя это на любителя, и зависит от размера шрифта. | |
| FixedCols | - количество фиксированных колонок. Они выделяются серым цветом, и всегда первые. Это свойство можно назвать заголовком строк. Практически не бывает необходимости делать более одной такой колонки, а иногда фиксированные колонки и вовсе не нужны. По умолчанию установлена одна колонка. | |
| FixedRows | - количество фиксированных строк. По умолчанию тоже одна, и работает так же, как и FixedCols. Как правило, эта строка служит заголовком колонок. | |
| GridLineWidth | - толщина разделительных линий. Попробуйте поставить ноль - линии исчезнут. Верните единицу. | |
| Options | - самое главное свойство компонента. Оно содержит много настроек, которые раскроются, если щелкнуть по плюсу слева от названия свойства. Эти настройки являются выключателями (True-разрешено, False-запрещено). Все настройки мы рассматривать не будем, их много, и большинство из них обычно не используется, кроме того, в справочнике Lazarus нет описаний этих настроек. Рассмотрим основные: | |
| goAlwaysShowEditor | - если True, то редактировать ячейку можно сразу при выделении. Если False, то для редактирования нужно нажать **<Enter>** или **<F2>**. |
| goColMoving | - можно ли мышью перемещать колонки на другое место. |
| goColSizing | - разрешено изменять ширину колонки мышью. |
| goFixedVertLine | - рисовать ли вертикальные линии у фиксированных ячеек? По умолчанию True. |
| goDrawFocusSelect | - разрешено выделять ячейку, которая находится в фокусе ввода. |
| goEditing | - можно ли редактировать сетку? То есть, вводить данные с клавиатуры. По умолчанию установлено False, то есть, редактировать нельзя. Игнорируется, если включен элемент goRowSelect. Если вы хотите, чтобы пользователь мог вводить в сетку данные, а обычно это нужно, этот параметр следует установить в True. |
| goFixedVertLine | - прорисовка вертикальных линий у фиксированных ячеек. |
| goHorzLine | - прорисовка горизонтальных линий у нефиксированных ячеек. |
| goRangeSelect | - разрешение выделять несколько ячеек. Не работает, если включен элемент goEdit. |
| goRowMoving | - можно ли мышью перемещать строки на другое место. |
| goRowSelect | - выделяется вся строка. Если равно False, то только одна ячейка. |
| goRowSizing | - разрешено изменять высоту строки мышью. |
| goTabs | - можно ли переключаться на другие ячейки с помощью клавиши **<Tab>**. |
| goThumbTracking | - разрешена ли прорисовка данных в ячейках при прокрутке. Если нет, то данные будут обновлены после прокрутки. |
| goVertLine | - прорисовка вертикальных линий у всех остальных (нефиксированных) ячеек. |
| RowCount | - количество строк в сетке. По умолчанию их 5, как и колонок. | |
| Row | - закрытое свойство, содержит индекс текущей (выделенной) строки. Строка может быть выделена целиком, или в ней может быть выделена только одна ячейка. Поскольку свойство закрыто, то в **Инспекторе объектов** вы его не найдете, однако при кодировании, программно, это свойство доступно. Вы можете узнать индекс текущей строки, но не сможете его изменить с помощью свойства Row. | |

Также сетка TStringGrid имеет пару полезных методов, с которыми тоже стоит познакомиться.

DeleteRow - метод удаляет строку, индекс которой указывается в параметре. Например:

StringGrid1.DeleteRow(StringGrid1.Row); //удалить текущую строку

StringGrid1.DeleteRow(0); //удалить первую

StringGrid1.DeleteRow(StringGrid1.RowCount-1); //удалить последнюю

SortColRow - метод выполняет сортировку данных в сетке как по строкам, так и по колонкам. Имеет два параметра:

SortColRow(IsColumn: Boolean; index: Integer);

Если IsColumn имеет значение True, то сортировка производится по колонке с индексом index. Если False, то по строке с индексом index. Как правило, данные требуется сортировать именно по колонкам, например, по колонке с фамилиями.

О чем еще следует знать? Данные сетки содержатся в отдельных ячейках. К ячейке можно обращаться по её индексу, как к элементу двухмерного массива, например:

s:= StringGrid1.Cells[1, 1]; //считали значение

StringGrid1.Cells[1, 1]:= 'Новое значение'; //записали значение

Как видно из примера, ячейки хранятся в списке ячеек Cells, при обращении к ячейке первым индексом является номер колонки, вторым - строки. Индексация начинается с нуля, поэтому самая верхняя левая ячейка будет иметь индекс [0, 0]. Если в сетке есть фиксированные строки и(или) колонки, именно они будут иметь нулевой индекс. То есть, если нужно указать в фиксированной строке заголовок третьей колонки, делать это надо так:

StringGrid1.Cells[2, 0]:= 'Заголовок';

Изменять ширину отдельных колонок нужно программно, используя свойство ColWidth:

ColWidth[<индекс\_колонки>]

Например, так:

StringGrid1.ColWidths[0]:= 120;

Таким образом, можно будет сделать колонки разной ширины, чего нельзя добиться при проектировании формы, в **Инспекторе объектов**.

Еще в ячейках можно использовать маски ввода для строк и колонок. Для этого нужно сгенерировать событие сетки OnGetEditMask, в котором и прописать маску. Как это делается, будет видно из кода примера ниже.

Доделаем сетку. Высоту строк (свойство DefaultRowHeight) установите в 18 пикселей. Разверните свойство Options и установите в True параметр goEditing, чтобы пользователь мог редактировать данные сетки. Количество фиксированных строк и колонок оставим по умолчанию - 1. Это будут заголовки столбцов и строк. Их нужно заполнить, для этого выделите форму и сгенерируйте для неё событие OnCreate. Код события такой:

procedure TForm1.FormCreate(Sender: TObject);

begin

//заполняем заголовки строк:

StringGrid1.Cells[0, 1]:= 'Иванов И.И.';

StringGrid1.Cells[0, 2]:= 'Петров П.П.';

StringGrid1.Cells[0, 3]:= 'Николаев Н.Н.';

StringGrid1.Cells[0, 4]:= 'Бонд Дж.';

//заполняем заголовки колонок:

StringGrid1.Cells[1, 0]:= 'Год рождения';

StringGrid1.Cells[2, 0]:= 'Место рождения';

StringGrid1.Cells[3, 0]:= 'Прописка';

StringGrid1.Cells[4, 0]:= 'Семейное положение';

//меняем ширину колонок:

StringGrid1.ColWidths[0]:= 120;

StringGrid1.ColWidths[1]:= 90;

StringGrid1.ColWidths[2]:= 150;

StringGrid1.ColWidths[3]:= 100;

StringGrid1.ColWidths[4]:= 120;

end;

Код, думаю, простой и понятный. Как только программа загрузится, сетка будет настроена. Теперь для колонки "**Год рождения**" создадим маску ввода, чтобы пользователю оставалось только вписать цифры. Выделите сетку, в **Инспекторе объектов** перейдите на вкладку **События** и сгенерируйте событие OnGetEditMask. Код события следующий:

procedure TForm1.StringGrid1GetEditMask(Sender: TObject; ACol, ARow: Integer;

var Value: string);

begin

if ACol=1 then Value:= '99.99.9999 г.';

end;

Как видите, в событии становятся доступными такие переменные, как ACol и ARow, в которых находится индекс соответственно, текущих колонки и строки. В переменную Value следует прописать текст маски, мы это делаем, только если текущей является вторая колонка (индексация с нуля):

if ACol=1 then Value:= '99.99.9999 г.';

Теперь запрограммируем кнопку, дав пользователю возможность добавлять новые фамилии. Сгенерируйте для кнопки событие OnClick, код события следующий:

procedure TForm1.Button1Click(Sender: TObject);

var

s: string;

begin

if InputQuery('Укажите новое имя',

'Впишите фамилию, имя и отчество в формате "Фамилия И.О.":',

s) then begin

StringGrid1.RowCount:= StringGrid1.RowCount + 1;

StringGrid1.Cells[0, StringGrid1.RowCount-1]:= s;

end;

end;

С помощью функции-запроса InputQuerry мы получаем в переменную s фамилию, имя и отчество. Затем добавляем еще одну строку:

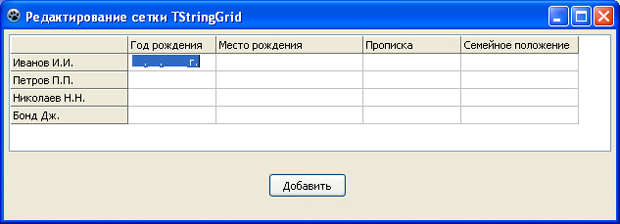
StringGrid1.RowCount:= StringGrid1.RowCount + 1;

В заключение, в последнюю строку первого (фиксированного) столбца мы вписываем эти новые ФИО:

StringGrid1.Cells[0, StringGrid1.RowCount-1]:= s;

Причем делается это только в том случае, если пользователь их указал, иначе ничего не происходит. Что же касается редактирования остальных данных, то пользователь будет делать это прямо в программе, в ячейках сетки.

Вот и все. Сохраните проект, запустите его на выполнения и опробуйте сетку в деле:



Приложение в работе