Практическое занятие 7. Отладка программы состоящей из нескольких модулей

Изучается структура модулей, их создание и подключение к проекту. На конкретном практическом примере рассматривается работа с модулями.

Цель занятия

Изучение внутренней структуры модулей, создание модуля и подключение его к проекту.

Модули

Модуль (англ. Unit) - *это автономно компилируемая программная единица, которая может включать в себя такие объекты программы, как типы, константы, переменные и подпрограммы (процедуры и функции).*

Каждый раз, когда мы создаем какое либо окно, для него автоматически создаются два файла: файл описаний **.lfm* и модуль **.pas*. В файле описаний в простом текстовом виде содержится описание формы - какие компоненты на ней находятся, как они настроены, какие параметры содержатся у свойств каждого компонента. Редактировать этот файл вручную крайне не рекомендуется, лучше всё сделать в Lazarus, в Инспекторе объектов и Редакторе форм, так вы гарантированно избежите ошибок в описании формы.

В файле модуля находится исходный код - то, что делает программа, когда пользователь взаимодействует с различными компонентами или с самой формой. Весь тот код, все процедуры и функции, которые мы вводили, находятся в модуле того или иного окна. Однако значение модулей этим не ограничивается. Модули необязательно должны быть связаны с окном, они могут существовать в проекте и сами по себе! Чаще всего это делают с двумя целями:

- Разработка приложений несколькими программистами (каждый пишет свой код, который потом собирается и компилируется в единый проект).
- Программист может собирать собственную библиотеку часто используемых функций и процедур, помещая их в независимые модули, которые затем он может подключить к любому своему проекту или даже поделиться ими с другими программистами.

Как вы понимаете, больше всего нас интересует вторая цель, с этим рано или поздно сталкивается любой практикующий программист. Кроме того, на просторах Интернета вы можете найти великое множество самых разнообразных инструментов, выполненных в виде модулей, которые авторы выкладывают в общий доступ, их можно скачать и использовать в своих программах.

Как-то раз один мой ученик спросил, не является ли дурным тоном использовать чужой код. Давным-давно, на заре программирования это было и впрямь не очень принято. Однако теперь все изменилось - программы стали гораздо больше, функциональней, простая программа содержит очень много кода. Например, мы с вами не раз применяли функцию `Length()`, которая возвращает длину строки в символах или длину массива в элементах. В принципе, мы могли бы поднапрячься, и самостоятельно реализовать эту функцию стандартными средствами Паскаля. Однако это было бы неоправданной тратой времени, ведь эта функция уже реализована и включена в библиотеку **Lazarus**! Я ответил тому ученику, что дурным тоном будет изобретать велосипед, то есть, впустую тратить время на разработку того, что давным-давно реализовано. Конечно, если вы будете использовать чужой код без ведома и разрешения автора, то это будет

плохо. Однако если автор сам выложил свой код, то почему бы им не воспользоваться? Разумеется, тут остается риск того, что автор - не очень хороший программист, и его код может "глючить" - работать с ошибками. В этом случае могу дать такой совет: ищите исходники на приличных, больших сайтах, которые не заброшены и часто обновляются.

Сейчас же нас больше интересует, как создать модуль самостоятельно, как поместить в него процедуры и функции, которые могут пригодиться нам в дальнейшем, в различных проектах. Мы реализуем модуль с "защитой от дураков" - с проверкой правильности вводимого пользователем числа. Причем числа могут быть целые беззнаковые (от 0 и выше), целые со знаком (могут быть отрицательными) и вещественные, которые все со знаком.

Структура модулей

Модуль в Паскале имеет следующую структуру:

```
Unit <имя модуля>;
interface
<интерфейсная часть> //открытая часть
implementation
<исполняемая часть (реализация)> //закрытая часть
initialization
<необязательная инициирующая часть>
finalization
<необязательная завершающая часть>
end.
```

Имя модуля

Имя модуля должно точно соответствовать имени файла этого модуля. То есть, если вы создаете модуль

unit MyUnit;

то и сохранять его нужно обязательно в файл **MyUnit.pas**. При этом старайтесь выбирать для модулей понятные имена, не соответствующие стандартным модулям. Имена стандартных подключаемых модулей вы можете видеть в разделе `uses` в **Редакторе кода**.

Совет: если при редактировании кода и удерживая нажатой клавишу **<Ctrl>** *вы подведете указатель мыши к имени какого либо подключаемого модуля, его имя превратится в гиперссылку, щелкнув по которой вы откроете этот модуль. Можете посмотреть, как реализуются профессиональные модули, заглянуть "во внутренности" процедур и функций.*

Интерфейсная часть

Интерфейсная часть - это открытая часть, начинается со служебного слова `interface`. Здесь можно объявлять глобальные типы, константы, переменные, функции и процедуры, которые будут доступны для всех программ и модулей, к которым вы подсоедините данный модуль. Здесь же при желании вы можете указать служебное слово uses, после которого перечислить те модули, которые хотите подключить к этому модулю. Пример (выполнять не нужно):

```
Unit MyUnit;
interface
uses windows;
const
   MyPI = 3.1415;
var
   MyGlobalVariable: Integer;
function MyGlobalFunc(a, b: string): boolean;
procedure MyClobalProc;
```

Здесь мы создаем модуль с именем `MyUnit`, который нужно сохранить в файл с именем **MyUnit.pas**. Далее, мы объявляем интерфейсную часть. Тут с помощью служебного слова `uses` (англ. *использовать*) мы указываем, что к этому модулю нужно еще подключить модуль `windows`, который входит в стандартную поставку **Lazarus**.

Далее, мы объявляем константу `MyPI`, которой сразу присваиваем значение, и переменную `MyGlobalVariable`. И константой, и переменной можно будет пользоваться извне - там, где будет подключаться этот модуль.

Далее мы объявляем функцию и процедуру. Обратите внимание: только объявляем! Само описание (код) этих функции и процедуры будет в разделе исполняемой части (или разделе реализации). В дальнейшем этими процедурой и функцией мы также сможем пользоваться извне.

Исполняемая часть

Исполняемая часть (раздел реализации) начинается со служебного слова `implementation`. Это закрытая часть. Тут описываются те процедуры и функции, которые были объявлены в интерфейсной части, причем заголовки подпрограмм должны в точности соответствовать предыдущему объявлению. Также тут можно объявлять типы, константы и переменные, а также и подпрограммы, которые будут видны только в этом модуле и не смогут использоваться извне. Пример:

```
implementation
uses MyOldUnit;
var
   MyLocalVariable: Real;

function MyGlobalFunc(a, b: string): boolean;
begin
   …
end;

procedure MyClobalProc;
begin
   …
end;
```

Как видно из примера, в исполняемой части также можно подключать внешние модули. Здесь же мы объявили переменную `MyLocalVariable`, она является глобальной внутри этого модуля, то есть, её можно использовать в функции и процедуре, однако извне она не будет видна. Далее у нас идет реализация (описание) объявленных ранее подпрограмм.

Инициирующая и завершающая части

Эти части модуля необязательны, их можно не указывать, как правило, их и не используют.

Инициирующая часть начинается служебным словом `initialization` и содержит код, который будет выполнен только один раз - до передачи управления в основную программу. Тут можно сделать какую то подготовительную работу - открыть нужный файл, присвоить значения каким то переменным и т.п.

Завершающая часть начинается служебным словом `finalization` и содержит код, который будет выполнен тоже один раз - при завершении работы основной программы. Здесь обычно освобождают занятые программой ресурсы, закрывают открытые файлы, сохраняют результаты работы и т.п.

Конец модуля

Модуль завершается служебным словом `end` с точкой в конце. Это единственный случай, когда после `end` ставится не точка с запятой, а точка. Весь дальнейший текст, если он есть, компилятором будет игнорирован. Кроме того, если дальнейший текст не соответствует синтаксису, компилятор выведет сообщение об ошибке, и не даст скомпилировать программу. Так что, если вам нужно указать какие то комментарии, например, с советами как пользоваться модулем, рекомендую делать их между фигурными скобками в самом начале модуля, даже до служебного слова `Unit`. Так обычно и делают. А вот после `end` с точкой уже ничего писать не нужно.

Создание модуля

Ну вот, с теорией покончили, приступим к практике. Как уже говорилось выше, сейчас мы с вами реализуем "защиту от дураков". Допустим, в программе у нас имеются компоненты `TEdit`, в которые пользователь должен вписать какое-то число - целое без знака, целое со знаком или вещественное, которое еще называют действительным числом. Должен то он должен, но кто может предположить, что он туда на самом деле введет? Может быть, слово, скобки, знак процента - пользователь непредсказуем! Как сказал один мудрый программист, "Если есть хоть малейшая возможность ввести в программу неправильные данные, пользователь эту возможность непременно найдет". Наша с вами задача - предусмотреть всё и лишить его этой возможности. Ведь нам в дальнейшем придется преобразовывать это число из текстового формата в настоящие числа, используя `StrToInt()`, `StrToFloat()` и т.п. функции. Если же пользователь введет ошибочные данные, то программа вызовет ошибку.

Поскольку делать такие проверки программисту приходится довольно часто, имеет смысл реализовать их в виде отдельного модуля, который затем можно будет подключить к любой программе. И этот модуль будет началом нашей библиотеки модулей, которую, как я надеюсь, вы будете постоянно пополнять как своим, так и чужим кодом.

Модуль - это текстовый файл с расширением **.pas*, который можно создать в любом текстовом редакторе. Однако будет удобней воспользоваться **Редактором кода** Lazarus - он подсвечивает синтаксис кода, выводит подсказки, при нажатии **<Ctrl+J>** выводит список с часто используемым командами. Так что для начала откроем **Lazars**. Если там загрузился предыдущий или новый проект, то командой меню **Файл->Закрыть** закройте его. Далее, выберите команду **Файл->Создать модуль**.

Сразу же появится заготовка модуля со следующим кодом:

```
unit Unit1;
{$mode objfpc}{$H+}
```

```
interface
uses
  Classes, SysUtils;
implementation
end.
```

Имя модуля по умолчанию `Unit1` - оно ни о чем нам не говорит, заменим его на `numbers`. Далее, идут директивы компилятору, затем интерфейсная часть с двумя подключенными модулями - их отключать не будем, поскольку там могут содержаться описания глобальных переменных, констант, подпрограмм, которые могут нам понадобиться. В частности, для проверки правильности ввода разделителя в вещественное число, мы используем системную переменную `DecimalSeparator`, которая описана в одном из этих модулей, и которую без него не получится использовать.

Целиком код модуля следующий (можете просто скопировать его отсюда):

```
//------ Начало модуля ---------
{В модуле описаны две функции, которые проверяют правильность вводимого
 пользователем символа. Функция TrueIntKeys проверяет целые числа и имеет
 параметры: key - введенный пользователем символ; str - строка в TEdit;
 sign - если True, то число со знаком, если False - без знака. Если пользо-
 ватель ввел правильный символ, функция его не изменяет, иначе возвращает
 символ #0 - то есть, ничего. Функция TrueFloatKeys аналогичным образом
 делает проверку вещественных чисел. Пример использования:
 Key:= TrueIntKeys(Key, Edit1.Text, True); //целое со знаком
 Key:= TrueIntKeys(Key, Edit1.Text, False); //целое без знака
 Key:= TrueFloatKeys(Key, Edit1.Text); //вещественное
}

unit Numbers;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

//проверка правильности символа в целом числе:
function TrueIntKeys(key:char; str:string; sign:boolean):char;
//проверка правильности символа в вещественном числе:
function TrueFloatKeys(key:char; str:string):char;

implementation

{проверка правильности символа в целом числе}
function TrueIntKeys(key:char; str:string; sign:boolean):char;
begin
  //сначала укажем, что возвращается тот же символ, что ввел пользователь:
  Result:= key;
  //далее делаем проверку на правильность символа. если символ не правильный,
  //мы его запретим:
  case key of
    //все числа разрешаем:
    '0'..'9': ;
    //backspace разрешаем:
    #8: ;
    //если знаковое, то разрешаем минус при условии, что минус - первый
    //символ в строке:
```

```
    '-': if sign and (Length(str) = 0) then Result:= key
         else Result:= #0; //если беззнаковое, или минус не первый, запрещаем
  //все остальные символы запрещаем:
  else Result:= #0;
  end; //case
end;

{проверка правильности символа в вещественном числе}
function TrueFloatKeys(key:char; str:string):char;
begin
  //сначала укажем, что возвращается тот же символ, что ввел пользователь:
  Result:= key;
  //далее делаем проверку на правильность символа. если символ не правильный,
  //мы его запретим:
  case key of
    //все числа разрешаем:
    '0'..'9': ;
    //backspace разрешаем:
    #8: ;
    //если разделителя еще нет - выводим правильный разделитель,
    //иначе ничего не выводим
    ',' , '.' : if Pos(DecimalSeparator, str)= 0 then
                  Result := DecimalSeparator
                else Result := #0;
    //разрешаем минус при условии, что минус - первый символ в строке:
    '-': if Length(str) = 0 then Result:= key
         else Result:= #0; //если минус не первый, запрещаем
  //все остальные символы запрещаем:
  else Result:= #0;
  end; //case
end;

end.
//------- Конец модуля ---------
```

Листинг . (html, txt)

Строки

```
//------ Начало модуля ---------
//------- Конец модуля ---------
```

копировать в модуль не обязательно, они только обозначают границы листинга этого кода.

Первым делом мы изменили имя модуля с `Unit1` на `Numbers`. Теперь этот модуль нужно сохранить в какую-нибудь временную папку, в файл **Numbers.pas**. Вместе с модулем будет сохраняться и проект, не обращайте на это внимание - проект нам не нужен, его потом можно будет удалить, оставив лишь файл с модулем.

Выберите команду **Файл->Сохранить**. Сначала будет запрос на сохранение проекта, его имя можно не менять, а папку выбрать временную, например, **C:\Temp**. Затем будет предложено сохранить модуль, причем Lazarus сам подставит имя `numbers`. Просто нажмите "**Сохранить**". Теперь откройте ту папку **Проводником** Windows или файловым менеджером, и увидите файл **numbers.pas** - это и есть наш модуль, остальные файлы в этой папке нам не нужны.

Прежде, чем двигаться дальше, давайте разберемся с кодом модуля.

В первоначальной заготовке модуля мы только поменяли его имя. Затем вернулись в самое начало, и в виде многострочного комментария описали работу с этим модулем. Делать это весьма полезно, так как модуль прослужит вам не год и не два - в дальнейшем вы можете просто забыть, как он был реализован. Чтобы вновь не изучать его исходный код, проще описать реализуемые тут инструменты и рекомендации по их использованию. А если вы планируете выложить свой модуль в Интернет, имеет смысл также добавить в комментарий строки

```
Автор: ФИО автора
Дата: Дата реализации
```

Затем в интерфейсной части мы объявили две функции:

```
function TrueIntKeys(key:char; str:string; sign:boolean):char;
function TrueFloatKeys(key:char; str:string):char;
```

Исходим вот из чего. У компонента `TEdit` имеется событие `OnKeyPress`, которое возникает **ДО** того, как введенный пользователем символ попадет в строку `Edit1.Text`. У события имеется переменная `Key`, которая и содержит этот символ. Как правило, в этом событии проверяют корректность введенного пользователем символа. Если символ правильный, то переменную оставляют без изменений. Если неправильный, его можно исправить - в коде мы сравниваем разделитель не с точкой или запятой, а с переменной `DecimalSeparator`, что гарантирует правильный разделитель, невзирая на то, точку нажал пользователь, или запятую. Ну а если уж символ совсем некорректный (например, буква в числе), то переменной `Key` в событии `OnKeyPress` можно присвоить `#0` - нулевой символ, то есть, ничто. В этом случае, пользователь может нажимать на неправильные клавиши хоть неделю, в `TEdit` эти символы не попадут. И, конечно же, в проверке обязательно надо учесть, что пользователю нужно предоставить возможность редактировать неправильно введенные символы. Так, клавиша **<Backspace>**, которая затирает символ слева от курсора, имеет код `#8` - эту клавишу нужно разрешить.

Реализацию самих функций мы сделали в разделе исполняемой части. Вначале переменной `Result` мы присвоили то значение, что уже содержится в параметре `key`. Ведь если символ правильный, то проще ничего не делать, чем каждый раз писать

```
Result:= key;
```

Затем для целого числа мы устраиваем следующую проверку:

```
  case key of
    //все числа разрешаем:
    '0'..'9': ;
    //backspace разрешаем:
    #8: ;
    //если знаковое, то разрешаем минус при условии, что минус - первый
    //символ в строке:
    '-': if sign and (Length(str) = 0) then Result:= key
         else Result:= #0; //если беззнаковое, или минус не первый, запрещаем
  //все остальные символы запрещаем:
  else Result:= #0;
  end; //case
```

Комментарии достаточно подробны, но все же разберем код "по косточкам". Переключатель `case` выполняет один из операторов, в зависимости от значения параметра `key`. Строка условий

```
'0'..'9': ;
```

означает, что если в `key` содержится символ любой цифры, от 0 до 9, то ничего делать не нужно, так как `Result` уже содержит символ, введенный пользователем, и этот символ корректен. Если бы мы, к примеру, хотели бы сделать проверку на латинский строчный символ, то указали бы диапазон `'a'..'z'`.

Далее, у нас проверка

```
#8: ;
```

Как мы уже выяснили, код `#8` имеет клавиша **<Backspace>**, которую мы тоже позволяем, поэтому ничего не делаем.

А вот дальше интересный код. Число может быть знаковым (параметр `sign=True`) - в этом случае нам нужно разрешить символ минуса. Однако все равно, мы должны проверить, что если это минус - он должен быть первым символом в строке, и что он единственный минус в строке. Согласитесь,

```
12--34
```

это неправильное целое число. Если же число беззнаковое, то мы и вовсе запрещаем этот минус. Всё это реализуется следующей проверкой:

```
'-': if sign and (Length(str) = 0) then Result:= key
     else Result:= #0; //если беззнаковое, или минус не первый, запрещаем
```

Этот оператор выполняется, если в параметре `key` у нас символ минуса. Условный оператор `if` выполняется, только если в параметре `sign` у нас `True`, и если строка пуста `(Length(str)=0)`. Согласитесь, если в строке уже что-то есть, то минус не будет первым символом! Если же число беззнаковое или вводимый символ - не первый, выполняется часть после `else` - функция возвращает код `#0`, то есть, результат будет такой, будто пользователь ничего и не нажимал.

И в заключение этой проверки мы также возвращаем код `#0`, сюда подпадут все остальные символы. В результате, пользователь не сможет ввести в `TEdit` никакого неправильного символа.

Проверка правильности вещественного числа в функции `TrueFloatKeys` такая же, как и для целого числа, с двумя исключениями: во-первых, все вещественные числа знаковые (могут быть отрицательными), поэтому параметр `sign` тут не нужен, и проверки на знаковость нет. Во-вторых, вещественное число может содержать разделитель целой и дробной части, а в строке не может быть двух разделителей. Это реализуется следующей проверкой:

```
    ',' , '.' : if Pos(DecimalSeparator, str)= 0 then
                  Result := DecimalSeparator
                else Result := #0;
```

Этот оператор выполняется, если в `key` точка или запятая. Функция

```
Pos(DecimalSeparator, str)
```

вернет ноль только в одном случае: если правильный разделитель, который хранится в системной переменной `DecimalSeparator`, не обнаружен в строке `str`. В этом случае мы возвращаем правильный разделитель, который может быть как точкой, так и запятой, в зависимости от языка операционной системы. Как видите, это универсальная проверка, не зависящая от языка, которая к тому же избавляет пользователя от необходимости следить за тем, точкой или запятой он отделяет целую и десятичную часть числа. Пользователю будет удобно пользоваться правой частью клавиатуры для ввода чисел. Если же в строке правильный разделитель уже есть, то мы возвращаем нулевой код, запрещая второй такой же разделитель.

Теперь о том, как использовать данный модуль в других проектах. Это можно сделать двумя способами. Первый заключается в том, что модуль нужно не только сохранить, но и скомпилировать. В этом случае, в папке с проектом появится папка **lib\i386-win32** (для Windows), в которой будут два файла модуля - **numbers.o** и **numbers.ppu**. Это - наш модуль в скомпилированном (исполняемом) виде. Теперь нужно найти папку, в которой в других папках хранятся библиотеки поставляемых с Lazarus различных модулей. Для Windows по умолчанию это папка

C:\lazarus\fpc\2.6.2\units\i386-win32

Адрес может быть и другой, если у вас иная версия FPC. Здесь мы можем создать свою папку, например, **MyModules** или **MyUnits**, куда и будем копировать наши модули в виде пары файлов, а затем подключать их к программам, включая имя модуля в раздел `uses`. Однако этот вариант не очень хорош - файлы-то двоичные, прочитать их простым текстовым редактором мы не сможем. Если таких файлов много, как узнать, где нужные нам инструменты, как дополнять эти модули новыми функциями и процедурами?

Я предлагаю другой способ. Создайте где-нибудь на диске, где хранятся ваши важные документы, папку с любым именем, куда вы будете собирать ваши (и чужие) модули, и скопируйте туда файл **numbers.pas** - наш модуль в исходном, не скомпилированном виде. А затем, если в каком то проекте нам нужно будет делать проверку на правильность числа, мы просто будем копировать **numbers.pas** в папку с этим проектом! Таким образом, он всегда будет доступен в текстовом варианте, по комментариям перед модулем мы легко сможем найти именно то, что нам нужно.

Итак, вы создали папку, которую будете использовать, как библиотеку ваших модулей, и скопировали туда **numbers.pas**? Отлично, теперь займемся самим проектом.

Пример включения модуля в проект

Откройте **Lazarus** с новым проектом. Как обычно, форму назовите **fMain**, в `Caption` пропишите `Проверка чисел`, сохраните проект в папку **7-01** под именем `CheckNum`, модулю главной формы, как обычно, дайте имя `Main`.

Теперь откройте ваш любимый файловый менеджер, с его помощью найдите наш модуль **numbers.pas** и скопируйте его в папку с проектом **7-01**. Далее, в **Редакторе кода** найдите раздел `uses` в интерфейсной части, поставьте запятую после последнего указанного включаемого модуля, добавьте туда имя

```
numbers:
uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, numbers;
```

Всё, наш модуль мы включили в проект, теперь можем реализовать проверку. Проект будет самый простой, с минимальным набором компонентов - только чтобы проверить работу нашего модуля. Установите на форму три компонента `TLabel`, их имена менять не нужно, а в `Caption` компонентов напишите соответственно, "`Целое со знаком:`", "`Целое без знака:`" и "`Вещественное:`". Правее меток установите три компонента `TEdit`, их имена также оставьте без изменений, в каждом `TEdit` очистите свойство `Text`. Ну и, наконец, в нижней части формы установите кнопку `TBitBtn` с вкладки **Additional**, в свойстве `Kind` которой выберите значение `bkClose`. Подравняйте компоненты так, чтобы у вас получилась примерно такая форма:

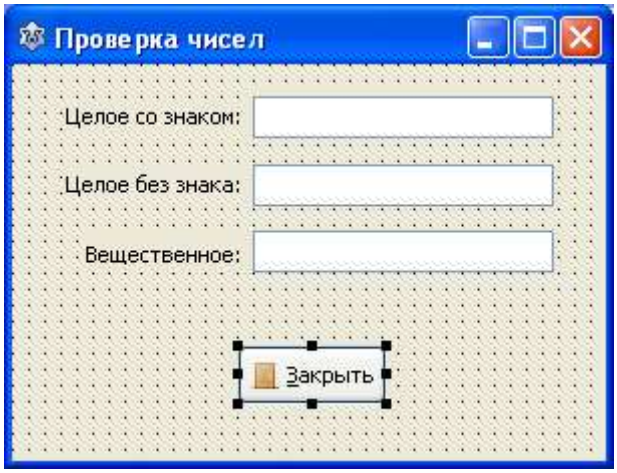

Рис. 7.1. Форма проверочного проекта

Теперь выделите `Edit1`, который предназначен для ввода целого числа со знаком. В Инспекторе объектов перейдите на вкладку **События** и сгенерируйте для компонента событие `OnKeyPress`. Его код:

```
procedure TfMain.Edit1KeyPress(Sender: TObject; var Key: char);
begin
  //проверка правильности вводимого символа в целое число со знаком:
  Key:= TrueIntKeys(key, Edit1.Text, true);
end;
```

Для `Edit2` код события `OnKeyPress` будет таким:

```
procedure TfMain.Edit2KeyPress(Sender: TObject; var Key: char);
begin
  //проверка правильности вводимого символа в целое число без знака:
  Key:= TrueIntKeys(key, Edit2.Text, false);
end;
```

И, наконец, для `Edit3` код события `OnKeyPress` будет таким:

```
procedure TfMain.Edit3KeyPress(Sender: TObject; var Key: char);
begin
  //проверка правильности вводимого символа в вещественное число:
  Key:= TrueFloatKeys(key, Edit3.Text);
end;
```

Всё, сохраните проект, скомпилируйте и запустите его на выполнение. Теперь, как бы вы ни старались, ввести некорректные значения в строки ввода у вас не получится:

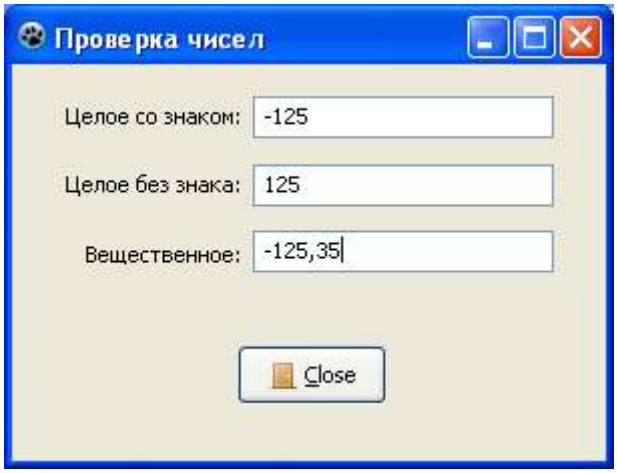

Рис. 7.2. Работающая программа

Откройте проект, который вы разрабатывали в Лабораторной работе № 3, включите в него разработанный модуль. Отладьте проект. Проведите тестирование модифицированного проекта. В качестве отчёта представьте описание выполненной работы.