9 Объектно-ориентированное программирование

Эта глава посвящена изучению объектно-ориентированного программирования (ООП). ООП – это методика разработки программы, в основе которой лежит понятие объекта как некоторой структуры, описывающей объект реального мира, его поведение и взаимодействие с другими объектами.

9.1 Основные понятия

Основой объектно-ориентированного программирования является объект. Он состоит из трех основных частей:

1. *Имя* (например, автомобиль);

2. *Состояние* или *переменные состояния* (например, марка автомобиля, цвет, масса, число мест и т. д.);

3. *Методы*, или операции, которые выполняют некоторые действия над объектами и определяют, как объект взаимодействует с окружающим миром.

Для работы с объектами во FreePascal введено понятие класса. *Класс* – сложная структура, включающая в себя описание данных, процедуры и функции, которые могут быть выполнены над объектом.

Классом называется составной тип данных, членами (элементами) которого являются функции и переменные (поля). В основу понятия «класс» положен тот факт, что "*над объектами можно совершать различные операции"*. Свойства объектов описываются с помощью переменных (полей) классов, а действия над объектами описываются с помощью подпрограмм, которые называются «методами класса». Объекты называются *экземплярами* класса.

Объектно-ориентированное программирование (ООП) – представляет собой технологию разработки программ с использованием объектов.

В объектно-ориентированных языках есть три основных понятия: инкапсуляция, наследование и полиморфизм. *Инкапсуляцией* называется объединение в классе данных и подпрограмм для их обработки. *Наследование* – это когда любой класс может быть порождён другим классом. Порождённый класс (наследник) автоматически наследует все поля, методы, свойства и события. *Полиморфизм* позволяет ис-

пользовать одинаковые имена для методов, входящих в различные классы.

Для объявления класса используется конструкция:

```
type
<название класса> = class(<имя класса родителя>)
<методы и переменные класса>
private
<поля и методы, доступные только
                                в пределах модуля>
protected
<поля и методы, доступные только
                                в классах-потомках>
public
   <поля и методы, доступные из других модулей>
published
   <поля и методы, видимые в инспекторе объектов>
end;
```

В качестве *имени класса* можно использовать любой допустимый в FreePascal идентификатор. *Имя класса родителя* — имя класса, наследником которого является данный класс, это необязательный параметр, если он не указывается, то это означает, что данный класс является наследником общего из предопределенного класса `TObject`.

Структуры отличаются от классов тем, что поля структуры доступны всегда. При использовании классов могут быть члены, доступные везде — публичные (описатель *public*), и приватные (описатель *private*), доступ к которым возможен только с помощью публичных методов. Это также относится и к методам класса.

Поля, свойства и методы секции *public* не имеют ограничений на видимость. Они доступны из других функций и методов объектов как в данном модуле, так и во всех прочих, ссылающихся на него. При обращении к публичным полям вне класса используется оператор . (точка).

Поля, свойства и методы, находящиеся в секции *private*, доступны только в методах класса и в функциях, содержащихся в том же модуле, что и описываемый класс. Это позволяет полностью скрыть детали внутренней реализации класса. Вызов приватных методов осуществляется из публичных.

Публикуемый (*published*) — это раздел, содержащий свойства, ко-

торые пользователь может устанавливать на этапе проектирования и они доступны в любом модуле.

Защищенный (*protected*) — это раздел, содержащий поля и методы, которые доступны внутри класса, а также любым его классам-потомкам, в том числе и в других модулях.

При программировании с использованием классов программист должен решить, какие члены и методы должны быть объявлены публичными, а какие приватными. Общим принципом является следующее: *"Чем меньше публичных данных о классе используется в программе, тем лучше"*. Уменьшение количества публичных членов и методов позволит минимизировать количество ошибок.

Поля могут быть любого типа, в том числе и классами. Объявление полей осуществляется так же, как и объявление обычных переменных:

```
поле1: тип_данных;
поле2: тип_данных;
...
```

Методы в классе объявляются так же, как и обычные подпрограммы:

```
function метод1 (список параметров): тип результата;
procedure метод2 (список параметров);
```

Описание процедур и функций, реализующих методы, помещается после слова `implementation` того модуля, где объявлен класс, и выглядит так:

```
function имя_класса.метод1(список параметров):
                                        тип результата;
begin
     тело функции;
end;
procedure имя_класса.метод2(список параметров);
begin
     тело процедуры;
end;
```

Объявление переменной типа `class` называется *созданием (инициализацией)* объекта (экземпляра класса). Экземпляр класса объявляется в блоке описания переменных:

```
var имя_переменной : имя_класса;
```

После описания переменной в программе можно обращаться к полям и методам класса аналогично обращению к полям структуры, используя оператор «.». Например:

```
имя_переменной.поле1:=выражение;
имя_переменной.метод1(список параметров);
...
```

Также можно использовать оператор With:

```
With имя_переменной do
begin
 поле1:=выражение;
 метод1(список параметров);
 ...
end;
```

В FreePascal имеется большое количество классов, с помощью которых описывается форма приложения и ее компоненты (кнопки, поля, флажки и т.п.). В процессе конструирования формы в текст программы автоматически добавляются программные объекты. Например, при добавлении на форму компонента формируется описание класса для этого компонента, а при создании подпрограмм обработки событий в описание класса добавляется объявление методов. Рассмотрим это на примере проекта с формой, на которой есть кнопка Button1.

```
unit Unit1;
interface
uses
Classes, SysUtils, LResources, Forms, Controls,
Graphics, Dialogs, StdCtrls;
type
{ TForm1 }
//объявление класса формы TForm1
TForm1 = class(TForm)
//объявление компонента кнопки Button1
Button1: TButton;
//объявление метода обработки события —
//щелчка по кнопке Button1
procedure Button1Click(Sender: TObject);
private
{ private declarations }
```

```
public
{ public declarations }
end;
var
   //описание переменной класса формы TForm1
    Form1: TForm1;
implementation
{ TForm1 }
//описание метода обработки события — щелчка по
кнопке Button1
procedure TForm1.Button1Click(Sender: TObject);
begin
    //Текст процедуры обработки события
end;
initialization
{$I unit1.lrs}
end.
```

Во Free Pascal класс (объект) –это *динамическая структура*. В отличие от статической она содержит не сами данные, а ссылку на них. Поэтому программист должен сам позаботиться о выделении памяти для этих данных.

Конструктор — это специальный метод, создающий и инициализирующий объект. Объявление конструктора имеет вид:

```
constructor Create;
```

Описывают конструктор так же, как и другие методы, после ключевого слова `implemention` того модуля, в котором объявлен класс.

```
constructor имя_класса.Create;
begin
 поле1:=выражение1;
 поле2:=выражение2;
 ...
inherited Create;
end;
```

В результате работы конструктора инициализируются все поля класса, при этом порядковым типам в качестве начальных значений задается 0, а строки задаются пустыми.

Деструктор – это специальный метод, уничтожающий объект и освобождающий занимаемую им память. Объявляется деструктор

следующим образом:

```
destructor Destroy;
```

Если в программе какой-либо объект больше не используется, то оператор

```
имя_переменной_типа_класс.free;
```

с помощью метода `free` вызывает деструктор и освобождает память, занимаемую полями объекта `имя_переменной_типа_класс`.

Рассмотрим все описанное на примере класса — комплексное число[78]. Назовем класс – `Tcomplex`, в классе будут члены класса: `x` – действительная часть комплексного числа, `y` – мнимая часть комплексного числа. Также в классе будут методы:

- конструктор `Create`, который будет записывать в действительную и мнимую части значение 0;
- `Modul()` – функция вычисления модуля комплексного числа;
- `Argument()` – функция вычисления аргумента комплексного числа;
- `ComplexToStr()` - функция, представляющая комплексное число в виде строки для вывода.

Создадим новый проект, на форму поместим кнопку `Button1`, два поля `Edit1` и `Edit2` для ввода действительной и мнимой частей, для вывода результатов разместим компонент `Memo1`. При щелчке по кнопке будет создаваться экземпляр класса «Комплексное число», затем будут вычисляться его модуль и аргумент. В компонент `Memo1` выведем результаты: число в алгебраической форме, его аргумент и модуль. Ниже приведем текст модуля с комментариями, который демонстрирует работу с этим классом. Результат работы программы приведен на рис. 9.1.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
```

78 Про комплексные числа можно прочитать на странице http://kvant.mccme.ru./1982/03/komplesnye_chisla.htm

```
 TForm1 = class(TForm)
 Button1: TButton;
 Edit1: TEdit;
 Edit2: TEdit;
 Label1: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Memo1: TMemo;
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
type
//Описание класса – комплексное число.
 TComplex = class
 private
     x: real;   //Действительная часть.
     y: real;   //Мнимая часть
 public
     constructor Create;   //Конструктор.
     //Метод вычисления модуля.
     function Modul():real;
     //Метод вычисления аргумента.
     function Argument():real;
     //Метод записи комплексного
     //числа в виде строки.
     function ComplexToStr(): String;
 end;
var
 Form1: TForm1;
//Объявление переменной
//типа класс «комплексное число».
 chislo: TComplex;
implementation
//описание конструктора
constructor TComplex.Create;
```

```
begin
 x:=0; y:=0;
 inherited Create;
end;
//Описание метода вычисления
//модуля комплексного числа.
function TComplex.Modul(): real;
begin
 modul:=sqrt(x*x+y*y);
end;
//Описание метода вычисления
//аргумента комплексного числа.
function TComplex.Argument(): real;
begin
 argument:=arctan(y/x)*180/pi;
end;
//Описание метода записи комплексного
//числа в виде строки.
function TComplex.ComplexToStr(): String;
begin
 if y>=0 then
     ComplexToStr:=FloatToStrF(x,ffFixed,5,2)+
          '+' + FloatTostrF(y,ffFixed,5,2)+ 'i'
 else
ComplexToStr:=FloatTostrF(x,ffFixed,5,2)+
             FloatTostrF(y,ffFixed,5,2)+ 'i'
end;
//Обработчик кнопки: создание экземпляра класса
//«Комплексное число», вычисление его модуля и
//аргумента, вывод числа в алгебраической
//форме, его аргумента и модуля.
procedure TForm1.Button1Click(Sender: TObject);
Var Str1: String;
begin
//Создание объекта (экземпляра класса)
//типа «комплексное число».
 chislo:=TComplex.Create;
//Ввод действительной и мнимой частей
```

```
//комплексного числа.
 chislo.x:=StrToFloat(Edit1.Text);
 chislo.y:=StrToFloat(Edit2.Text);
 Str1:='Kompleksnoe chislo '+
                    chislo.ComplexToStr();
//Вывод на форму в поле Memo
        //построчно комплексного числа,
 Memo1.Lines.Add(Str1) ;
        //его модуля
 Str1:='Modul chisla '+
     FloatToStrF(chislo.Modul(),ffFixed, 5, 2);
 Memo1.Lines.Add(Str1) ;
        //и аргумента.
 Str1:='Argument chisla '+
 FloatToStrF(chislo.Argument(),ffFixed, 5, 2);
 Memo1.Lines.Add(Str1) ;
//Уничтожение объекта.
 chislo.Free;
end;
initialization
  {$I unit1.lrs}
end.
```

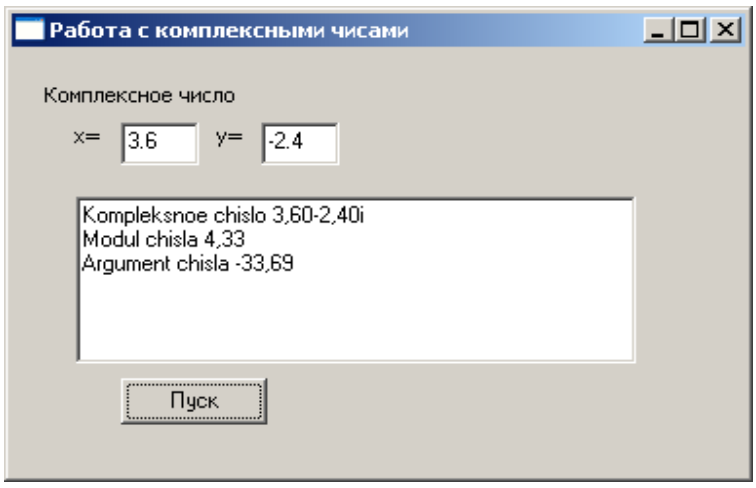

Рисунок 9.1: Результаты программы работы с классом «Комплексное число»

В этом примере был написан конструктор для класса «комплексное число» без параметров. В Free Pascal можно написать конструктор с параметрами, который принимает входные значения и инициализирует поля класса этими значениями. Перепишем предыдущий

пример следующим образом. Действительную и мнимую части будем считывать из полей ввода формы и передавать в конструктор для инициализации объекта типа комплексное число. Листинг программы приведен ниже.

```
   unit Unit1;
   {$mode objfpc}{$H+}
   interface
   uses
    Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
   type
    { TForm1 }
    TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
    procedure Memo1Change(Sender: TObject);
   private
    { private declarations }
   public
    { public declarations }
   end;
   type
   TComplex = class
    private x, y: real;
    public
   //Объявление конструктора.
        constructor Create(a,b:real);
        function Modul():real;
        function Argument():real;
        function ComplexToStr(): String;
   end;
   var
```

```
 Form1: TForm1;
 chislo: TComplex;
implementation
//Конструктор, который получает
//в качестве входных параметров
//два вещественных числа и записывает
//их в действительную и мнимую части
//комплексного числа.
constructor TComplex.Create(a,b:real);
begin
     x:=a;   y:=b;
     inherited Create;
end;
function TComplex.Modul(): real;
begin
 modul:=sqrt(x*x+y*y);
end;
function TComplex.Argument(): real;
begin
 argument:=arctan(y/x)*180/pi;
end;
function TComplex.ComplexToStr(): String;
begin
 if y>=0 then
     ComplexToStr:=FloatToStrF(x,ffFixed,5,2)+
         '+' + FloatTostrF(y,ffFixed,5,2)+ 'i'
 else
     ComplexToStr:=FloatTostrF(x,ffFixed,5,2)+
                 FloatTostrF(y,ffFixed,5,2)+ 'i'
end;
procedure TForm1.Button1Click(Sender: TObject);
Var  Str1 : String;
     x1, x2 : real;
begin
 x1:=StrToFloat(Edit1.Text);
 x2:=StrToFloat(Edit2.Text);
 chislo:=TComplex.Create(x1,x2);
 chislo.x:=StrToFloat(Edit1.Text);
```

```
 chislo.y:=StrToFloat(Edit2.Text);
 Str1:='Kompleksnoe chislo '+
                        chislo.ComplexToStr() ;
 Memo1.Lines.Add(Str1) ;
 Str1:='Modul chisla '+
     FloatToStrF(chislo.Modul(),ffFixed, 5, 2);
 Memo1.Lines.Add(Str1) ;
 Str1:='Argument chisla '+
 FloatToStrF(chislo.Argument(),ffFixed, 5, 2);
 Memo1.Lines.Add(Str1) ;
 chislo.Free;
end;
initialization
 {$I unit1.lrs}
end.
```

9.2 Инкапсуляция

Инкапсуляция — один из важнейших механизмов объектно-ориентированного программирования (наряду с наследованием и полиформизмом). Класс представляет собой единство трех сущностей – полей, свойств и методов, что и является инкапсуляцией. Инкапсуляция позволяет создавать класс как нечто целостное, имеющее определённую функциональность. Например, класс `TForm` содержит в себе (инкапсулирует) все необходимое, чтобы создать диалоговое окно.

Основная идея инкапсуляции – защитить поля от несанкционированного доступа. Поэтому целесообразно поля объявлять в разделе `private`. Прямой доступ к полям объекта: чтение и обновление их содержимого должно производиться посредством вызова соответствующих методов. В FreePascal для этого служат свойства класса.

Свойства - это специальный механизм классов, регулирующий доступ к полям. Свойства объявляются с помощью зарезервированных слов `property`, `read` и `write`. Обычно свойство связано с некоторым полем и указывает те методы класса, которые должны использоваться при записи в это поле или при чтении из него. Синтаксис объявления свойств следующий:

```
propety имя_1: тип read имя_чтения write имя_2
```

Зарезервированное слово `read` описывает метод чтения свойств объекта, а слово `write` описывает метод записи свойств объекта. `Имя_1` и `имя_2` – соответственно имена методов, обеспечивающих чтение или запись свойства.

Если необходимо, чтобы свойство было доступно только для чтения или только для записи, следует опустить соответственно часть `write` или `read`.

Рассмотрим следующий пример. Создадим класс – многоугольник, имя класса `TPolygon`. Полями класса будут:

- `K` – количество сторон многоугольника;
- `p` – массив, в котором будут храниться длины сторон многоугольника.

Методами класса будут:

- конструктор `Create`, обнуляющий элементы массива `p`;
- `Perimetr()` - функция вычисления периметра фигуры;
- `Show()` - функция формирования сведений о фигуре (количество сторон и периметр);
- `Set_Input()` - функция проверки исходных данных.

Расположим на форме кнопку и метку. При щелчке по кнопке появляется окно ввода количества сторон многоугольника. Если количество сторон введено корректно, то инициализируется объект «Многоугольник» с количеством сторон, равным введенному, в противном случае количество сторон многоугольника принимается равным по умолчанию 50. После этого вычисляется периметр фигуры и результаты выводятся на форму в метке `Label1`.

Ниже приведен листинг программы с комментариями, результаты работы программы можно увидеть на рис. 9.2.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: TButton;
```

```
 Label1: TLabel;
 procedure Button1Click(Sender: TObject);
 private
     { private declarations }
 public
     { public declarations }
end;
type
//Объявление класса многоугольник Tpolygon.
TPolygon = class
//Закрытые поля.
Private
 K : integer;
 p : array of real;
//Открытые методы.
Public
 constructor Create;       //Конструктор.
 //Метод вычисления периметра.
 function Perimetr():real;
//Метод формирования сведений.
 function Show():String;
Protected      //Защищенные методы.
 //Процедура проверки данных.
 procedure Set_Input(m:integer);
Published
//Объявление свойства n.
//Свойство n оперирует полем K.
//В описании свойства после слова read
//стоит имя поля — K. Это значит, что
//функция чтения отсутствует и пользователь
//может читать непосредственно значение поля.
//Ссылка на функцию  Set_Input после
//зарезервированного слова  write означает,
//что с помощью этой функции в поле K будут
//записываться новые значения.
Property n: integer read K write Set_Input;
end;
var
```

```
 Form1: TForm1;
//Объявление переменной типа
//класс многоугольник.
 Figure: TPolygon;
implementation
//Описание конструктора.
constructor TPolygon.Create;
var i:integer;
begin
 K:=50;//Присваивание начальных значений полям.
//Выделение памяти под массив p.
 SetLength(p,K);
 for i:=0 to K-1 do p[i]:=0;
 inherited Create;
end;
//Функция вычисления периметра.
function TPolygon.Perimetr():real;
var Sum:real; i:integer;
begin
 Sum:=0;
 for i:=0 to K-1 do Sum:=Sum+p[i];
 Perimetr:=Sum;
end;
//Метод формирования сведений о фигуре.
function TPolygon.Show():String;
begin
 Show:='Многоугольник с количеством сторон '+
            IntToStr(K)+chr(13)+'Периметр = '+
                        FloatToStr(Perimetr())
end;
//Метод записи данных в поле K.
procedure TPolygon.Set_Input(m:integer);
begin
//Если введенное значение положительное число,
//то записать его в поле K,
//иначе вернуться к начальному значению.
 if m>1 then K:=m  else K:=50;
end;
```

```
{TForm1 }
//Обработка события.
procedure TForm1.Button1Click(Sender: TObject);
var  i, m:integer;
 s:string;
begin
//Ввод количества сторон многоугольника.
s:=InputBox('Ввод','Введите количество сторон
                          многоугольника','6');
 Val(s,m);
 //Инициализация объекта.
 Figure:=TPolygon.Create;
 with Figure do
 begin
//Метод проверки исходных данных.
     Set_Input(m);
//Формирование массива случайных чисел.
     for i:=0 to K-1 do p[i]:=random(50);
//Обращение к методу вычисления периметра.
         s:=Show();
 end;
 //Вывод результатов в окно формы.
 Label1.Caption:= s;
end;
initialization
{$I unit1.lrs}
end.
```

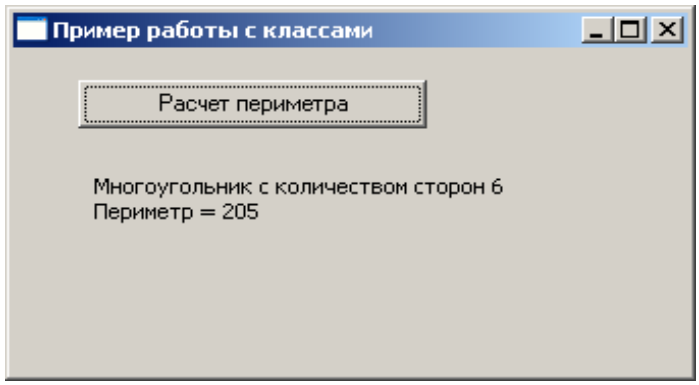

Рисунок 9.2: Результат работы программы с классом многоугольник

9.3 Наследование и полиформизм

Второй основополагающей составляющей объектно-ориентированного программирования является *наследование*. Смысл наследования заключается в следующем: если нужно создать новый класс, лишь немного отличающийся от старого, то нет необходимости в переписывании заново уже существующих полей и методов. В этом случае объявляется новый класс, который является наследником уже имеющегося, и добавляются к нему новые поля, методы и свойства. При создании новый класс является *наследником* членов ранее определенного *базового* класса (родителя). Класс-наследник является *производным* от базового, причем он сам может выступать в качестве базового класса для вновь создаваемых классов.

В Object Pascal все классы являются потомками класса `TObject`. Поэтому если вы создаете дочерний класс прямо от класса `TObject`, то в определении его можно не упоминать.

Производный класс наследует от класса-предка поля и методы; если имеет место совпадение имен методов, то говорят, что они перегружаются. В зависимости от того, какие действия происходят при вызове, методы делятся на следующие группы:

- статические методы;
- виртуальные методы;
- динамические методы.

По умолчанию все методы *статические*. Эти методы полностью перегружаются в классах-потомках при их переопределении. При этом можно полностью изменить объявление метода. Если обращаться к такому методу у объекта базового класса, то будет работать метод класса-родителя. Если обращаться к методу у производного класса, то будет работать новый метод.

Виртуальные и *динамические* методы имеют в базовом и производном классах те же имена и типы. В классах-наследниках эти методы перегружены. В зависимости от того, с каким классом работают, соответственно и вызывается метод этого класса.

Основная разница между виртуальными и динамическими методами — в способе их вызова. Информация о виртуальных методах хранится в таблице виртуальных методов VMT. В VMT хранятся виртуальные методы данного класса и всех его предков. При создании потомка класса вся VMT предка переносится в потомок и там к ней

добавляются новые методы. Поиск нужного метода занимает мало времени, так как класс имеет всю информацию о своих виртуальных методах. Динамические методы не дублируются в таблице динамических методов DMT потомка. DMT класса содержит только методы, объявленные в этом классе. При вызове динамического метода сначала осуществляется поиск в DMT данного класса, и если метод не найден – то в DMT предка класса и т.д. Таким образом использование виртуальных методов требует большего расхода памяти из-за необходимости хранения массивных VMT всех классов, зато они вызываются быстрее.

Изменяя алгоритм того или иного метода в производных классах, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо перегрузить его в потомке, т. е. объявить в наследнике одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два одноименных метода, имеющие разную алгоритмическую основу. Это называется *полиморфизмом* объектов.

Виртуальные и динамические методы объявляются так же, как и статические, только в конце описания метода добавляются служебные слова `virtual` или `dynamic`:

```
type
 имя_родителя =class
     ...
     метод;  virtual;
     ...
end;
```

Чтобы перегрузить в классе-наследнике виртуальный метод, нужно после его объявления написать ключевое слово `override`:

```
type
 имя_наследника=class (имя_родителя)
     ...
     метод;  override;
     ...
end;
```

Рассмотрим наследование в классах на следующем примере. Создадим базовый класс `Ttriangle` (*треугольник*) с полями класса –

координаты вершин треугольника. В классе будут следующие методы:

- `Proverka()` – метод проверки существования треугольника (если 3 точки лежат на одной прямой, то треугольник не существует);
- `Perimetr()` – метод вычисления периметра треугольника;
- `Square()` – - метод вычисления площади;
- Методы вычисления длин сторон `a(), b(), c()`;
- `Set_Tr()` – метод получения координат;
- `Show()` – метод формирования сведений о треугольнике.

На основе этого класса создадим производный класс `R_TTriangle` (равносторонний треугольник), который наследует все поля и методы базового класса, но методы проверки и формирования сведений о фигуре перегружаются по другому алгоритму.

На форме поместим метку, кнопку и по 6 компонентов типа `TEdit` для ввода координат вершин для двух треугольников. После щелчка по кнопке создаются два объекта типа «Треугольник» и «Равносторонний треугольник», вычисляются периметр и площадь каждого треугольника, и результаты выводятся ниже в компоненты `Label1 Label2`. Далее приведен листинг программы с комментариями.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
          Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
TForm1 = class(TForm)
 Button1: Tbutton;  //кнопка Расчет
 //массивы x1, y1 – координаты вершин
//1-го треугольника частный случай)
 //массивы x1, y1 – координаты вершин
//2-го треугольника (равностороннего)
 Edit1: Tedit;  //для ввода координаты x1[1]
 Edit10: TEdit; //для ввода координаты y2[2]
 Edit11: TEdit; //для ввода координаты x2[3]
```

```
 Edit12: TEdit; //для ввода координаты y2[3]
 Edit2: TEdit;  //для ввода координаты y1[1]
 Edit3: TEdit;  //для ввода координаты x2[2]
 Edit4: TEdit;  //для ввода координаты y1[2]
 Edit5: TEdit;  //для ввода координаты x1[3]
 Edit6: TEdit;  //для ввода координаты y1[3]
 Edit7: TEdit;  //для ввода координаты x2[1]
 Edit8: TEdit;  //для ввода координаты y2[1]
 Edit9: TEdit;  //для ввода координаты x2[2]
 Label1: TLabel;
 Label10: TLabel;
 Label11: TLabel;
 Label12: TLabel;
 Label13: TLabel;
 Label14: TLabel;
 Label15: TLabel;
 Label16: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Label4: TLabel;
 Label5: TLabel;
 Label6: TLabel;
 Label7: TLabel;
 Label8: TLabel;
 Label9: TLabel;
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
//Объявление базового класса «треугольник».
type
 TTriangle=class
 Private
//Массивы, в которых хранятся координаты
//вершин треугольника.
    x, y:array[0..2] of real;
```

```
Public
constructor Create;//конструктор
//Метод получения исходных данных.
procedure  Set_Tr(a,b:array of real);
//Методы вычисления сторон треугольника.
function a():real;
function b():real;
function c():real;
//Виртуальный метод проверки существования
//треугольника, который будет перегружен
//в производном классе.
function Proverka():boolean;      virtual;
//Метод вычисления периметра.
function Perimetr():real;
//Метод вычисления площади.
function Square():real;
//Виртуальный метод формирования
//сведений о треугольнике.
function Show():string;        virtual;
end;
//Объявление производного
//класса «равносторонний треугольник».
type
R_TTriangle=class(TTriangle)
public
//Перегружаемые методы проверки, что
//треугольник является равносторонним,
//и формирования сведений о треугольнике.
function Proverka():boolean; override;
function Show():string;        override;
end;
var
 Form1: TForm1;
//Объявление переменной типа
//класс «треугольник».
 Figura1: Ttriangle;
//Объявление переменной типа
//класс «равносторонний треугольник».
```

```
 Figura2: R_TTriangle;
implementation
//Конструктор, который обнуляет
//массивы координат.
constructor TTriangle.Create;
var i:integer;
begin
 for i:=0 to 2 do
 begin
     x[i]:=0;  y[i]:=0;
 end;
end;
//Метод получения координат вершин.
procedure  TTriangle.Set_Tr(a,b:array of real);
var i:integer;
begin
 for i:=0 to 2 do
 begin
     x[i]:=a[i];  y[i]:=b[i];
 end;
end;
//Методы вычисления сторон треугольника a, b, c
function TTriangle.a():real;
begin
 a:=sqrt(sqr(x[1]-x[0])+sqr(y[1]-y[0]));
end;
function TTriangle.b():real;
begin
 b:=sqrt(sqr(x[2]-x[1])+sqr(y[2]-y[1]));
end;
function TTriangle.c():real;
begin
 c:=sqrt(sqr(x[0]-x[2])+sqr(y[0]-y[2]));
end;
//Методы вычисления периметра треугольника.
function TTriangle.Perimetr():real;
begin
 Perimetr:=a()+b()+c();
```

```
   end;
   //Функции вычисления площади треугольника.
   function TTriangle.Square():real;
   var p:real;
   begin
    p:=Perimetr()/2;   //полупериметр
    Squire:=sqrt((p-a())*(p-b())*(p-c()));
   end;
   //Метод проверки существования треугольника:
   //если в уравнение прямой, проходящей через
   //две точки, подставить координаты 3-й точки
   //и при этом получится равенство, значит, три
   //точки лежат на одной прямой и построение
   //треугольника невозможно.
   function TTriangle.Proverka():boolean;
   begin
    if (x[0]-x[1])/(x[0]-x[2])=
                  (y[0]-y[1])/(y[0]-y[2]) then
       Proverka:=false
    else Proverka:=true
   end;
   //Метод формирования строки –
   //сведений о треугольнике.
   function TTriangle.Show():string;
   begin
   //Если треугольник существует,
   //то формируем строку сведений о треугольнике.
    if Proverka() then
       Show:='Tr'+chr(13)+'a='+
       FloatToStrF(a(),ffFixed,5,2)+
chr(13)+'b='+FloatToStrF(b(),ffFixed,5,2)+
chr(13)+'c='+FloatToStrF(c(),ffFixed,5,2)+
chr(13)+'P='+FloatToStrF(Perimetr(),ffFixed,5,2)+
chr(13)+'S='+FloatToStrF(Square(),ffFixed,5,2)
       else
       Show:='Not Triangle';
   end;
```

```
//Метод проверки существования
//равностороннего треугольника.
function R_TTriangle.Proverka():boolean;
begin
 if (a()=b()) and(b()=c()) then
     Proverka:=true
                        else
     Proverka:=false
end;
//Метод формирования сведений
//о равностороннем треугольнике.
function R_TTriangle.Show():string;
begin
//Если треугольник равносторонний,
//то формируем строку сведений.
 if Proverka()=true then
Show:='Tr'+chr(13)+'a='+
FloatToStrF(a(),ffFixed,5,2)+
chr(13)+'P='+
FloatToStrF(Perimetr(),ffFixed,5,2)+chr(13)
+'S='+FloatToStrF(Square(),ffFixed,5,2)
 else
     Show:='Not R_Triangle';
end;
{ TForm1 }
procedure TForm1.Button1Click(Sender: Tobject);
//Массивы x1, y1 – координаты треугольника.
//Массивы x2, y2 – координаты
//равностороннего треугольника
 var x1, y1, x2, y2 :array[1..3] of real;
 s:string;
begin
//Чтение координат треугольников
//из полей ввода диалогового окна.
 x1[1]:=StrToFloat(Edit1.Text);
 y1[1]:=StrToFloat(Edit2.Text);
 x1[2]:=StrToFloat(Edit3.Text);
 y1[2]:=StrToFloat(Edit4.Text);
```

```
 x1[3]:=StrToFloat(Edit5.Text);
 y1[3]:=StrToFloat(Edit6.Text);
 x2[1]:=StrToFloat(Edit7.Text);
 y2[1]:=StrToFloat(Edit8.Text);
 x2[2]:=StrToFloat(Edit9.Text);
 y2[2]:=StrToFloat(Edit10.Text);
 x2[3]:=StrToFloat(Edit11.Text);
 y2[3]:=StrToFloat(Edit12.Text);
//Инициализация объекта класса треугольник.
 Figura1:=TTriangle.Create;
//Инициализация объекта класса
//равносторонний треугольник.
 Figura2:=R_TTriangle.Create;
 Figura1.Set_Tr(x1,y1);
 Figura2.Set_Tr(x2,y2);
//Вызов методов формирования
//сведений и вывод строки на форму.
 s:=Figura1.Show();
 Label15.Caption:= S;
 s:=Figura2.Show();
 Label16.Caption:= S;
//уничтожение объектов
 Figura1.Free;
 Figura2.Free;
end;
initialization
 {$I unit1.lrs}
end.
```

Результаты работы программы представлены на рис. 9.3.

Абстрактный метод – это виртуальный или динамический метод, реализация которого не определена в том классе, где он объявлен. Предполагается, что этот метод будет перегружен в классе-наследнике. Вызывают метод только в тех классах, где он перезагружен. Объявляется абстрактный метод при помощи служебного слова `abstract` после слов `virtual` или `dynamic`, например:

```
        метод1;    virtual;        abstract;
```

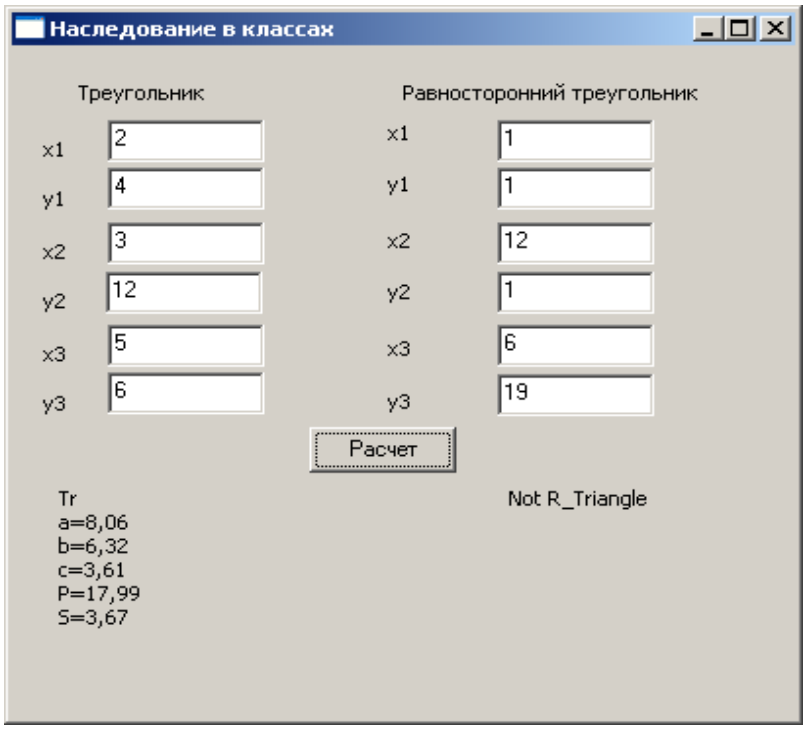

Рисунок 9.3: Результаты работы программы расчета параметров треугольников

Рассмотрим следующий пример. Создадим базовый класс `TFigure` (фигура). На основании этого класса можно построить производные классы для реальных фигур (окружность, четырехугольник и т.д.). В нашем примере рассмотрим два класса-наследника `TCircle` (окружность) и `TRectangle` (прямоугольник). На форму поместим кнопку. При щелчке по кнопке специализируются 2 объекта типа «Окружность» и «Прямоугольник», рассчитываются параметры фигур : периметр (длина окружности), площадь, результаты выводятся в компоненты `Label1` и `label2`. Ниже приведен листинг программы.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
         Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: TButton;
```

```
 Label1: TLabel;
 Label2: TLabel;
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
//Объявление базового класса «фигура».
type
 TFigure=class
private
 n:integer; //Количество сторон фигуры.
 p:array of real;   //Массив длин сторон фигуры.
public
//Абстрактный конструктор
//в каждом производном классе будет
//перегружаться.
constructor Create;        virtual; abstract;
//Метод вычисления периметра
function Perimetr():real;
//Абстрактный метод вычисления площади,в каждом
//производном классе будет перегружаться
//реальным методом.
function Square():real;virtual; abstract;
//Абстрактный метод формирования
//сведений о фигуре
//в каждом производном классе
//будет перегружаться.
function Show():string;virtual; abstract;
end;
//Объявление производного класса «окружность».
type
TCircle=class(TFigure)
public
constructor Create;        override;
function Perimetr():real;
function Square():real;    override;
```

```
function Show():string;     override;
end;
//Объявление производного
//класса «прямоугольник».
type
TRectangle=class(TFigure)
public
constructor Create;     override;
function Square():real;    override;
function Show():string;     override;
end;
var
 Form1: Tform1;
//Объект типа класса окружность.
 Figura1: Tcircle;
//Объект типа класса прямоугольник.
 Figura2:TRectangle;
implementation
//Описание метода вычисления
//периметра для базового класса.
function  TFigure.Perimetr():real;
var     i:integer;
    s:real;
begin
 s:=0;
 for i:=0 to n-1 do
    s:=s+p[i];
 Perimetr:=s;
end;
//Описание конструктора в классе окружность
//перезагрузка абстрактного
//родительского конструктора.
constructor TCircle.Create;
begin
//Количество сторон для окружности 1.
 n:=1;
//Выделяем память под 1 элемент массива.
 SetLength(p,n);
```

```
 p[0]:=5;   //Сторона - радиус окружности.
end;
//Перезагрузка метода вычисления периметра.
function  TCircle.Perimetr():real;
begin
//Вычисление длины окружности.
 Perimetr:=2*Pi*p[0];
end;
//Перезагрузка метода вычисления площади
function  TCircle.Square():real;
begin
 Square:=Pi*sqr(p[0]);
end;

//Описание метода формирования
//строки сведений об окружности.
//Перезагрузка родительского
//абстрактного метода.
function TCircle.Show():string;
begin
 Show:='Circle'+chr(13)+'r='+
 FloatToStr(p[0])+chr(13)+'P='+
 FloatToStr(Perimetr())+
 chr(13)+'S='+FloatToStr(Square());
end;
//Описание конструктора в классе прямоугольник.
//перезагрузка абстрактного
//родительского конструктора.
constructor TRectangle.Create;
begin
 n:=2;   //Количество сторон — две.
//Выделение памяти под два элемента.
 SetLength(p,n);
 p[0]:=4; p[1]:=2;  //Длины сторон.
End;
//Перезагрузка абстрактного
//родительского метода,
//вычисления площади фигуры.
```

```
function  TRectangle.Square():real;
begin
 Square:=p[0]*p[1];
end;

//Описание метода формирования
//сведений о прямоугольнике.
//Перезагрузка родительского
//абстрактного метода.
function TRectangle.Show():string;
begin
 Show:='Rectangle'+chr(13)+'a='+
 FloatToStr(p[0])+'  b='+
 FloatToStr(p[1])+
chr(13)+'P='+FloatToStr(Perimetr())+
 chr(13)+'S='+FloatToStr(Square());
end;
{ TForm1 }
procedure TForm1.Button1Click(Sender: TObject);
var s:string;
begin
//Инициализация объекта типа окружность.
 Figura1:=TCircle.Create;
//Инициализация объекта типа прямоугольник.
 Figura2:=TRectangle.Create;
//Формирование сведений о фигурах
//и вывод результатов на форму.
 s:=Figura1.Show() ;
 label1.Caption:= s ;
 s:=Figura2.Show() ;
 label2.Caption:= s ;
 Figura1.Free;
 Figura2.Free;
end;
initialization
{$I unit1.lrs}
end.
```

9.4 Перегрузка операций

Во Free Pascal можно перегрузить не только функции, но и операции, например, можно запрограммировать, чтобы операция * при работе с матрицами осуществляла умножение матриц, а при работе с комплексными числами – умножение комплексных чисел.

Для этого в программе нужно написать специальную функцию – метод. Объявление метода перегрузки записывается после объявления класса и выглядит так:

```
operator symbols(параметры:тип)имя_результата:тип;
```

где `operator` – служебное слово;

`symbols` – символ перегружаемой операции;

`параметры` – имена переменных, участвующих в перегрузке оператора;

Описывают метод перегрузки так же, как и другие методы, после ключевого слова `implemention` того модуля, в котором объявлен класс.

Рассмотрим несколько примеров.

Задача 9.1. Создать класс работы с комплексными числами, в котором перегрузить операции сложения и вычитания.

На форме разместим четыре компонента типа `TEdit` для ввода действительной и мнимой частей двух комплексных чисел. Также создадим компонент типа `TMemo` для вывода результатов и кнопку. При щелчке по кнопке создаются четыре объекта типа «комплексное число». Первое и второе числа инициализируются введенными данными. Третье формируется как результат сложения первых двух, а четвертое — как разность этих же чисел. Текст программы с комментариями приведен ниже, на рис. 9.4 показано окно работы программы.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes, SysUtils, LResources, Forms,
          Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: Tbutton;
```

```
//Для ввода действительной части первого числа.
 Edit1: Tedit;
//Для ввода мнимой части первого числа.
 Edit2: TEdit;
//Для ввода действительной части второго числа.
 Edit3: TEdit;
//Для ввода мнимой части второго числа.
 Edit4: TEdit;
 Label1: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Label4: TLabel;
 Label5: TLabel;
 Label6: TLabel;
 Memo1: Tmemo;  //Поле для вывода результатов.
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
//Объявление класса – комплексное число.
type
 TComplex = class
 private
     x: real;   //Действительная часть.
     y: real;   //Мнимая часть.
 public
     constructor Create;   //Конструктор.
//Функция вычисления модуля комплексного числа.
 function Modul():real;
//Функция вычисления аргумента
//комплексного числа;
 function Argument():real;
//Функция, представляющая комплексное число
//в виде строки для вывода.
 function ComplexToStr(): String;
end;
```

```
//Методы перегрузки оператора «+»
//с комплексными числами.
 operator +(const a, b: TComplex)r: TComplex;
//Методы перегрузки оператора «-»
//с комплексными числами.
 operator -(const a, b: TComplex)r: TComplex;
var
 Form1: TForm1;
//Объявление переменных типа комплексное число.
 chislo1: Tcomplex;
 chislo2: TComplex;
 chislo3: TComplex;
 chislo4: TComplex;
implementation
//Описание конструктора.
constructor TComplex.Create;
begin
     x:=0; y:=0;
     inherited Create;
end;
function TComplex.Modul(): real;
begin
     modul:=sqrt(x*x+y*y);
end;
function TComplex.Argument(): real;
begin
     argument:=arctan(y/x)*180/pi;
end;
//Методы перегрузки оператора «+».
//переменные a и b – комплексные числа,
//которые складываются,
//r – результирующее комплексное число
operator +(const a, b: TComplex)r: TComplex;
begin
 r:=TComplex.Create; //Инициализация объекта.
//Действительная часть нового комплексного
//числа формируется как результат сложения
//действительных частей первого и второго
```

```
//операндов. Мнимая часть нового комплексного
//числа формируется как результат сложения
//мнимых частей первого и второго операндов.
//Операнды - переменные типа комплексное
//число, передаваемые в метод.
 r.x:=a.x+b.x;
 r.y:=a.y+b.y;
end;
//Методы перегрузки оператора «-».
operator -(const a, b: TComplex)r: TComplex;
begin
 r:=TComplex.Create; //Инициализация объекта.
//Действительная часть нового комплексного
//числа формируется как разность действительных
//частей первого и второго операндов. Мнимая
//часть нового комплексного числа формируется
//как разность мнимых частей первого и второго
//операндов. Операнды - переменные типа
//комплексное число, передаваемые в метод.
 r.x:=a.x-b.x;
 r.y:=a.y-b.y;
end;
function TComplex.ComplexToStr(): String;
begin
 if y>=0 then
 ComplexToStr:=FloatToStrF(x,ffFixed,5,2)+'+'+
     FloatTostrF(y,ffFixed,5,2)+ 'i'
         else
 ComplexToStr:=FloatTostrF(x,ffFixed,5,2)+
     FloatTostrF(y,ffFixed,5,2)+ 'i'
end;
procedure TForm1.Button1Click(Sender: TObject);
Var Str1 : String;
begin
//Инициализация объектов.
//chislo1, chislo2 – исходные комплексные
//числа, вводимые с формы.
//chislo3 – Комплексное число,
```

```
//получаемое сложением двух исходных чисел.
//chislo4 – Комплексное число, получаемое
//вычитанием второго числа из первого.
 chislo1:=TComplex.Create;
 chislo2:=TComplex.Create;
 chislo3:=TComplex.Create;
 chislo4:=TComplex.Create;
//Чтение действительной и мнимой частей
//двух исходных комплексных чисел
//из полей ввода формы.
 chislo1.x:=StrToFloat(Edit1.Text);
 chislo1.y:=StrToFloat(Edit2.Text);
 chislo2.x:=StrToFloat(Edit3.Text);
 chislo3.y:=StrToFloat(Edit4.Text);
//Для сложения двух комплексных чисел можно
//использовать операцию +,так как она
//перегружена для этого класса.
 chislo3:=chislo1+chislo2;
//Для вычитания двух комплексных чисел
//можно использовать операцию -,так как она
//перегружена для этого класса.
 chislo4:=chislo1-chislo2;
//Результат сложения двух чисел преобразуется
//в строку для вывода в поле Memo1.
 Str1:='chislo1+chislo2 '+
                chislo3.ComplexToStr() ;
 Memo1.Lines.Add(Str1) ;
//Значение модуля преобразуется
//в строку для вывода в поле Memo1.
 Str1:='Modul chisla '+
  FloatToStrF(chislo3.Modul(),ffFixed, 5, 2);
Memo1.Lines.Add(Str1);
//Значение аргумента преобразуется в строку
//для вывода в поле Memo1.
Str1:='Argument chisla '+
FloatToStrF(chislo3.Argument(), ffFixed, 5, 2);
Memo1.Lines.Add(Str1) ;
//Результат разности двух чисел преобразуется
```

```
//в строку для вывода в поле Memo1.
Str1:='chislo1-chislo2 '+
                    chislo4.ComplexToStr() ;
Memo1.Lines.Add(Str1) ;
Str1:='Modul chisla '+
  FloatToStrF(chislo4.Modul(),ffFixed, 5, 2);
Memo1.Lines.Add(Str1) ;
Str1:='Argument chisla '+
FloatToStrF(chislo4.Argument(), ffFixed,5,2);
Memo1.Lines.Add(Str1) ;
end;
initialization
 {$I unit1.lrs}end.
```

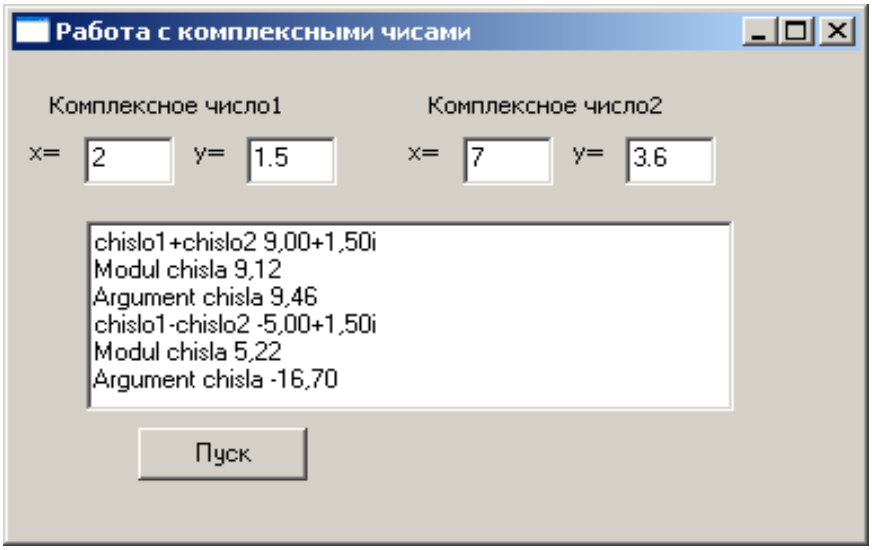

Рисунок 9.4: Пример работы программы к задаче 9.1

Задача 9.2. Создать класс работы с квадратными матрицами, в котором перегрузить операции сложения и умножения. В классе создать метод проверки, что матрица является единичной.

В классе созданы два метода перегрузки операции сложения и два метода перегрузки операции умножения. Это методы сложения и умножения матриц, а также методы добавления к матрице числа и умножения матрицы на число.

На форму поместим кнопку и 8 компонентов типа `TLabel` для вывода результатов. При щелчке по кнопке создаются 6 объектов типа «матрица», все они заполняются случайными числами. Третья матрица затем рассчитывается как сумма первых двух. Четвертая матрица

— результат умножения первой матрицы на вторую. Пятая матрица получается из первой путем добавления к ней числа 10, а шестая — умножением первой матрицы на число 5. Также первая и вторая матрицы проверяются, не являются ли они единичными. Текст программы с комментариями приведен далее.

```
   unit Unit1;
   {$mode objfpc}{$H+}
   interface
   uses
    Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
   type
    { TForm1 }
    TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Memo1Change(Sender: TObject);
   private
    { private declarations }
   public
    { public declarations }
   end;
   //Объявление класса матрица.
   type
    TMatrix = class
    private
        //элементы матрицы
        x: array [0..5,0..5]of real;
    public
        constructor Create;       //Конструктор.
   //Метод формирования строки,
   //выводящей элементы матрицы.
```

```
    function MatrixToStr(): String;
//Функция проверки, является ли
//матрица единичной.
 function Pr_Edin(): boolean;
end;
//Методы перегрузки операции «+» -
//сложение двух матриц.
 operator +(const a, b: TMatrix)r: Tmatrix;
//Методы перегрузки операции «+» -
//добавление к матрице вещественного числа.
operator+(const a:TMatrix;b:real)r:TMatrix;
//Метоы перегрузки операции «*» -
//умножение двух матриц.
 operator *(const a, b: TMatrix)r: Tmatrix;
//Методы перегрузки операции «*» -
//умножение матрицы на вещественное число.
 operator *(const a:TMatrix;b:real)r: TMatrix;
var
 Form1: TForm1;
 matr1: TMatrix;
 matr2: TMatrix;
 matr3: TMatrix;
 matr4: TMatrix;
 matr5: TMatrix;
 matr6: TMatrix;
implementation
//Конструктор в классе матрица, который
//заполняет матрицу случайными целыми числами.
constructor TMatrix.Create;
var i, j : integer;
begin
 for i:=0 to 4 do
     for j:=0 to 4 do
         x[i,j]:=Random(10);
 inherited Create;
end;
//Функция, проверяющая,
//является ли матрица единичной.
```

```
   function TMatrix.Pr_Edin():boolean;
   var i, j : integer;
   begin
   //Предполагаем, что матрица единичная.
    Result:=true;
    for i:=0 to 4 do
        for j:=0 to 4 do
   //Если на главной диагонали элемент не 1
   //или вне главной диагонали элемент не 0,
if ((i=j)and(x[i,j]<>1))or((i<>j)and(x[i,j]<>0))
then
    begin
   //то матрица не является единичной.
            Result:=false;
            break;  //выход из цикла
    end;
   end;
   //Метод перегрузки оператора «+» -
   //сложение двух матриц
   operator +(const a, b: TMatrix)r: TMatrix;
   var i, j: integer;
   begin
    r:=TMatrix.Create;
    for i:=0 to 4 do
        for j:=0 to 4 do
            r.x[i,j]:=a.x[i,i]+b.x[i,j];
    end;
   //Методы перегрузки оператора «*» -
   //перемножение двух матриц.
   operator *(const a, b: TMatrix)r: TMatrix;
   var  i, j, k: integer;
    s: real;
   begin
    r:=TMatrix.Create;
    for i:=0 to 4 do
        for j:=0 to 4 do
            begin
                r.x[i,j]:=0;
```

```
            for k:=0 to 4 do
                r.x[i,j]:=
                r.x[i,j]+a.x[i,k]*b.x[k,j];
        end;
end;
//Методы перегрузки оператора «+» -
//сложение матрицы и вещественного числа.
operator +(const a: TMatrix;b:real)r: TMatrix;
var i, j: integer;
begin
 r:=TMatrix.Create;
 for i:=0 to 4 do
     for j:=0 to 4 do
         r.x[i,j]:=a.x[i,j]+b;
end;
//Методы перегрузки оператора «*» -
//умножение матрицы на вещественное число
operator *(const a: TMatrix;b:real)r: TMatrix;
var i, j: integer;
begin
 r:=TMatrix.Create;
 for i:=0 to 4 do
     for j:=0 to 4 do
         r.x[i,j]:=a.x[i,j]*b;
end;
//Методы формирования строки
//с элементами матрицы.
function TMatrix.MatrixToStr(): String;
var  s:string;
     i, j :integer;
begin
 s:='';//Вначале строка пустая
 for i:=0 to 4 do
     begin
         for j:=0 to 4 do
//Добавление к s строки, которая представляет
//собой элемент матрицы и пробел.
     s:=s+FloatTostrF(x[i,j],ffFixed,5,2)+'  ';
```

```
//Формирование новой строки матрицы.
        s:=s+chr(13);
 end;
 MatrixToStr:=s;
end;
//Обработка кнопки Пуск:
//создание экземпляров класса Матрица,
//вычисление результирующих матриц путем
//сложения, умножения исходных матриц,
//умножение матрицы на число
procedure TForm1.Button1Click(Sender: TObject);
Var Str1 : String;
begin
//Инициализация объектов:
//matr1, matr2 – исходные матрицы,
//заполняемые случайными числами;
//matr3 – матрица, получаемая
//сложением двух исходных матриц;
//matr4 – матрица, получаемая умножением
//двух исходных матриц;
//matr5 – матрица, получаемая добавлением
//к первой матрице числа 10;
//matr6 – матрица, получаемая умножением
//первой матрицы на число 5.
 matr1:=TMatrix.Create;
 matr2:=TMatrix.Create;
 matr3:=TMatrix.Create;
 matr4:=TMatrix.Create;
 matr5:=TMatrix.Create;
 matr6:=TMatrix.Create;
//Вычисление матриц matr3, matr4, matr5, matr6
//с помощью перегрузки операторов.
 matr3:=matr1+matr2;
 matr4:=matr1*matr2;
 matr5:=matr1+10;
 matr6:=matr1*5;
//Формирование строки вывода и ее вывод
//в метку Label1 матрицы matr1.
```

```
 Str1:='Matrix1 '+chr(13)+matr1.MatrixToStr() ;
 Label1.Caption:= Str1;
//Проверка, является ли матрица matr1
//единичной,и вывод соответствующего сообщения.
 if matr1.Pr_Edin()=true then
     Str1:='Matrix1 edinichnaya'
 else
     Str1:='Matrix1 ne edinichnaya';
 Label5.Caption:= Str1 ;
//Проверка, является ли матрица matr2
//единичной,и вывод соответствующего сообщения.
 if matr2.Pr_Edin()=true then
     Str1:='Matrix2 edinichnaya'
 else
     Str1:='Matrix2 ne edinichnaya';
 Label6.Caption:= Str1 ;
 Str1:='Matrix2 '+chr(13)+matr2.MatrixToStr() ;
 Label2.Caption:= Str1 ;
//Вывод элементов матрицы matr3=matr1+matr2.
 Str1:='Matrix1+Matrix2 '+
            chr(13)+matr3.MatrixToStr() ;
 Label3.Caption:= Str1 ;
//Вывод элементов матрицы matr4=matr1*matr2.
 Str1:='Matrix1*Matrix2 '+
            chr(13)+matr4.MatrixToStr() ;
 Label4.Caption:= Str1 ;
//Вывод элементов матрицы matr5=matr1+10.
 Str1:='Matrix1+10 '+
            chr(13)+matr5.MatrixToStr() ;
 Label7.Caption:= Str1 ;
//Вывод элементов матрицы matr5=matr1*5.
 Str1:='Matrix1*5 '+chr(13)+
                  matr6.MatrixToStr() ;
 Label8.Caption:= Str1 ;
end;
initialization
 {$I unit1.lrs}
end.
```

Результат работы программы показан на рис. 9.5 .

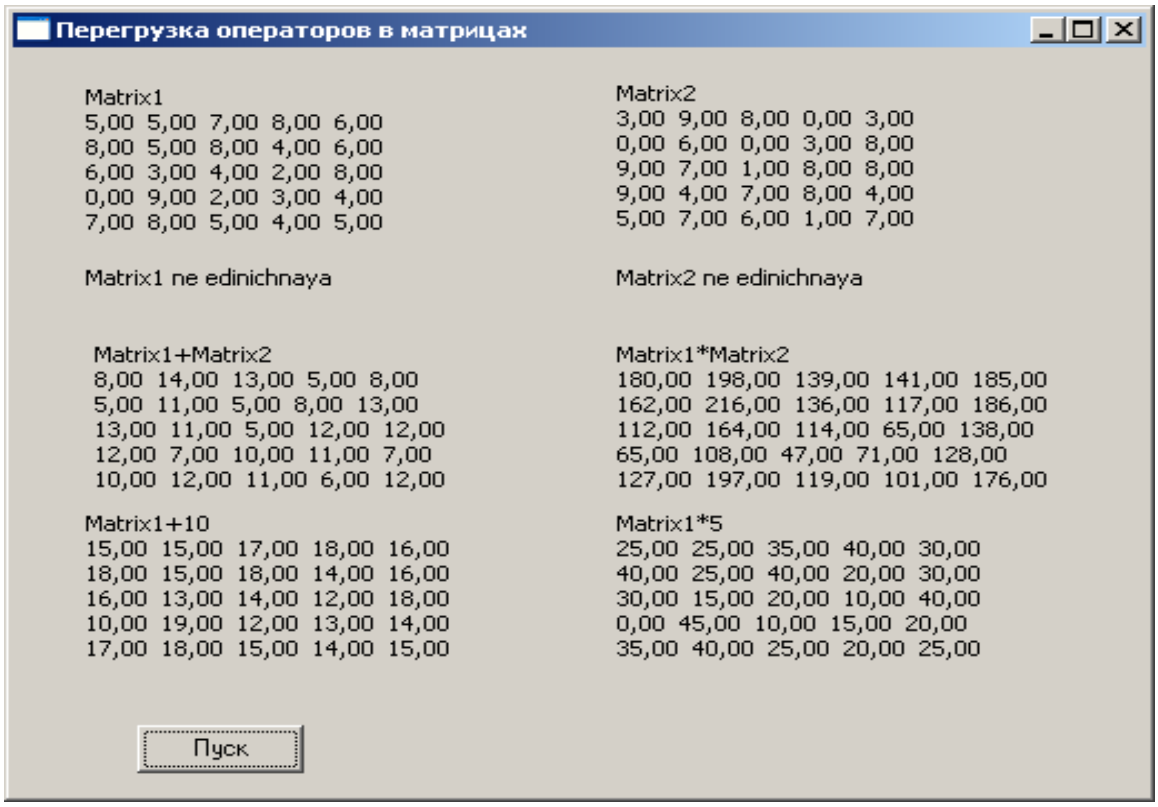

Рисунок 9.5: Результаты работы программы задачи 9.2

ЗАДАЧА 9.3. Создать класс – обыкновенная дробь, в котором перегрузить операции < и >.

На форму поместим четыре компонента `TEdit` для ввода числителей и знаменателей двух обыкновенных дробей и компонент `Memo1` для вывода результатов. Также разместим кнопку, при щелчке по которой инициализируются два объекта типа «обыкновенная дробь», затем они сравниваются и результаты выводятся в Memo1. Ниже приведен листинг программы с комментариями, на рис. 9.6 показаны результаты работы программы.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
 Classes,    SysUtils,    LResources,    Forms,
Controls, Graphics, Dialogs, StdCtrls;
type
 { TForm1 }
 TForm1 = class(TForm)
 Button1: Tbutton;
```

```
//Поле для ввода числителя первой дроби.
 Edit1: Tedit;
//Поле для ввода знаменателя первой дроби.
 Edit2: Tedit;
//Поле для ввода числителя второй дроби.
 Edit3: Tedit;
//Поле для ввода знаменателя второй дроби.
 Edit4: Tedit;
 Label1: TLabel;
 Label2: TLabel;
 Label3: TLabel;
 Label4: TLabel;
 Memo1: Tmemo;  //Поле для вывода результатов.
 procedure Button1Click(Sender: TObject);
private
 { private declarations }
public
 { public declarations }
end;
//Объявление класса – обыкновенная дробь.
type
 TOb_Drob = class
 private
     Ch: integer;       //Числитель.
     Zn: integer;       //Знаменатель.
 public
     //Конструктор.
     constructor Create(a, b:integer);
 //Метод формирования строки для вывода дроби.
     function DrobToStr(): String;
end;
//Метод перегрузки оператора «<».
operator <(const a, b: TOb_Drob)r: boolean;
//Метод перегрузки оператора «>».
operator >(const a, b: TOb_Drob)r: boolean;
var
 Form1: Tform1;
//Объявление переменных типа класс
```

```
//обыкновенная дробь.
 d1, d2: TOb_Drob;
implementation
//Конструктор.
constructor TOb_drob.Create(a, b:integer);
begin
 Ch:=a; Zn:=b;
 inherited Create;
end;
//Метод перегрузки оператора «<» -
//сравнение обыкновенных дробей.
operator <(const a, b: TOb_Drob)r: boolean;
begin
 if a.Ch*b.Zn <b.Ch * a.Zn then
        r:=true
 else r:=false ;
end;
//Метод перегрузки оператора «>» -
//сравнение обыкновенных дробей.
operator >(const a, b: TOb_Drob)r: boolean;
begin
 if a.Ch*b.Zn > b.Ch * a.Zn then
        r:=true
 else r:=false ;
end;
//Метод формирования строки
//для вывода дроби на форму.
function TOb_Drob.DrobToStr(): String;
begin
 if Zn<>0 then
     if Ch*Zn>0 then
     DrobToStr:=IntToStr(Ch)+'/'+
                                IntToStr(Zn)
     else
         DrobToStr:='-'+IntToStr(abs(Ch))+'/'+
                            IntTostr(abs(Zn))
 else
        DrobToStr:='Dividing by a zero'
```

```
end;
procedure TForm1.Button1Click(Sender: TObject);
Var  Str1 : String;
     a, b: integer;
begin
//Чтение данных из полей ввода формы:
 a:=StrToInt(Edit1.Text);  //числитель,
 b:=StrToInt(Edit2.Text);  //знаменатель.
//Инициализация первой дроби.
 d1:=TOb_Drob.Create(a,b);
//Чтение данных из полей ввода формы.
 a:=StrToInt(Edit3.Text);
 b:=StrToInt(Edit4.Text);
//Инициализация второй дроби.
 d2:=TOb_Drob.Create(a,b);
//Формирование строки вывода
//и добавление ее в поле Memo1.
//Вывод исходной дроби d1.
 Str1:='Drob 1 '+d1.DrobToStr() ;
 Memo1.Lines.Add(Str1) ;
//Вывод исходной дроби d2.
 Str1:='Drob2 '+d2.DrobToStr() ;
 Memo1.Lines.Add(Str1) ;
//сравнение дробей с помощью перегрузки
//операторов < и > и вывод сообщения.
 if d1<d2   then
     Str1:='Drob1 < Drob2'
            else
         if d1>d2   then
             Str1:='Drob1 > Drob2'
                 else
             Str1:='Drob1 = Drob2';
 Memo1.Lines.Add(Str1) ;
end;
initialization
  {$I unit1.lrs}
end.
```

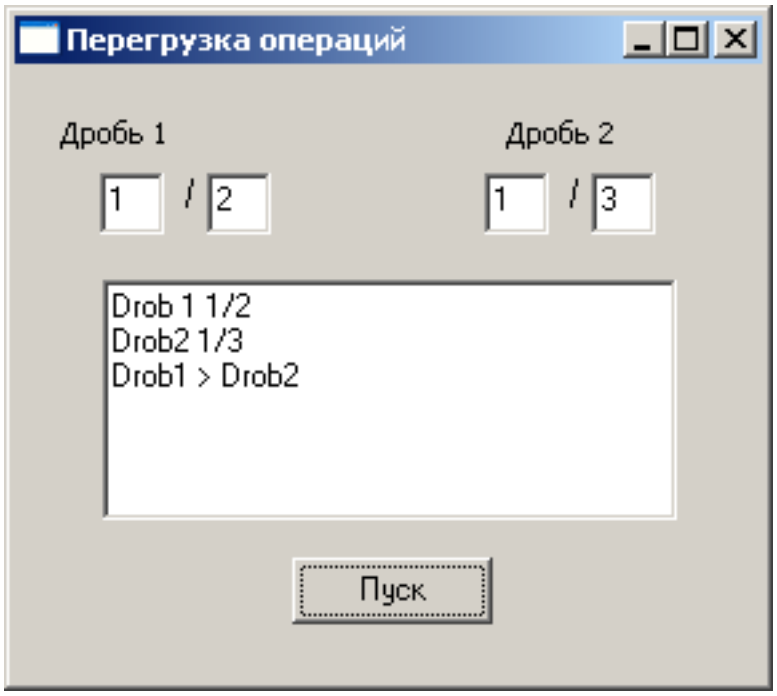

Рисунок 9.6: Результаты работы программы задачи 9.3

9.5 Задачи для самостоятельного решения

1. Создать класс *комплексное число* в алгебраической форме `z=x+y·i`, поля класса – действительная (`x`) и мнимая (`y`) части числа. Методы класса: вычисление корня комплексного числа, вывод комплексного числа. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение комплексных чисел.

2. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Метод класса: вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, проверку, является ли одна матрица обратной другой ($A \cdot A^{-1} = E$).

3. Создать класс *вектор на плоскости*, поля класса – координаты вектора. Методы класса: вычисление направляющих косинусов вектора, вывод всех характеристик вектора. В классе предусмотреть методы перегрузки операций: сложение, скалярное и векторное произведения векторов.

4. Создать класс *обыкновенная дробь*, поля класса – числитель и знаменатель. Методы класса: сокращение дроби, вывод дроби. В классе предусмотреть методы перегрузки операций: сложение, вычитание,

деление и умножение дробей.

5. Создать класс *квадрат*, член класса – длина стороны. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – периметр, площадь, диагональ. Создать производный класс – *куб*, добавить в класс метод определения объема фигуры, перегрузить методы расчета площади и вывода сведений о фигуре.

6. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Методы класса: вычисление суммы всех элементов матрицы, вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, умножение матрицы на число.

7. Создать класс *прямая,* поля класса – координаты двух точек (`x1, y1`) и (`x2, y2`). Метод класса: вывод уравнения прямой вида `y= ax + b`. В классе предусмотреть методы перегрузки операций: проверка параллельности двух прямых, определение угла между двумя прямыми.

8. Создать класс *комплексное число* в тригонометрической форме $a = \rho(\cos\varphi + i\sin\varphi)$, поля класса – модуль (ρ) и аргумент (φ) числа. Методы класса: возведение числа в степень, вывод комплексного числа в алгебраической и тригонометрической формах. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение комплексных чисел.

9. Создать класс *вектор на плоскости*, поля класса – координаты вектора. Методы класса: вычисление длины вектора, вывод характеристик вектора. В классе предусмотреть методы перегрузки операций: сложение, скалярное и векторное произведения векторов.

10. Создать класс *обыкновенная дробь*, поля класса – числитель и знаменатель. Методы класса: определение обратной дроби, вывод дроби. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение дробей.

11. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Методы класса: проверка, является ли матрица верхнетреугольной или нижнетреугольной, вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, умножение матрицы на число.

12. Создать класс *треугольник*, члены класса – длины 3-х сторон. Предусмотреть в классе методы проверки существования треугольни-

ка, вычисления и вывода сведений о фигуре – длины сторон, углы, периметр, площадь. Создать производный класс – *равнобедренный треугольник*, предусмотреть в классе проверку, является ли треугольник равнобедренным.

13. Создать класс *комплексное число* в показательной форме $a = \rho e^{i\varphi}$, поля класса – модуль (ρ) и аргумент (φ) числа. Методы класса: вывод комплексного числа в алгебраической, тригонометрической и показательной формах. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение комплексных чисел.

14. Создать класс *прямая,* поля класса – коэффициенты уравнения `y= ax + b`. Методы класса: вывод уравнения прямой, определение точек пересечения с осями. В классе предусмотреть методы перегрузки операций: проверка перпендикулярности двух прямых, определение угла между двумя прямыми.

15. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Методы класса: проверки, является ли матрица диагональной или нулевой, вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, добавление к матрице числа.

16. Создать класс *треугольник*, члены класса – координаты 3-х точек. Предусмотреть в классе методы проверки существования треугольника, вычисления и вывода сведений о фигуре – длины сторон, углы, периметр, площадь. Создать производный класс – *прямоугольный треугольник*, предусмотреть в классе проверку, является ли треугольник прямоугольным.

17. Создать класс *комплексное число* в тригонометрической форме $a = \rho(\cos\varphi + i\sin\varphi)$, поля класса – модуль (ρ) и аргумент (φ) числа. Методы класса: извлечение корня из числа, вывод комплексного числа в алгебраической и тригонометрической формах. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение комплексных чисел.

18. Создать класс *обыкновенная дробь*, поля класса – числитель и знаменатель. Методы класса: возведение дроби в степень, вывод дроби. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение дробей.

19. Создать класс *треугольник*, члены класса – длины 3-х сторон.

Предусмотреть в классе методы проверки существования треугольника, вычисления и вывода сведений о фигуре – длины сторон, углы, периметр, площадь. Создать производный класс – *равносторонний треугольник*, предусмотреть в классе перегрузку метода проверки существования равностороннего треугольника.

20. Создать класс *комплексное число* в алгебраической форме `z=x+ y ·i`, поля класса – действительная (`x`) и мнимая (`y`) части числа. Методы класса: вычисление модуля и аргумента комплексного числа, вывод комплексного числа. В классе предусмотреть методы перегрузки операций: сложение, вычитание комплексных чисел, проверки сопряженности двух комплексных чисел.

21. Создать класс *окружность*, член класса – радиус *R*. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – площадь, длина окружности. Создать производный класс – *круглый прямой цилиндр с высотой h***,** добавить в класс метод определения объема фигуры, перегрузить методы расчета площади и вывода сведений о фигуре.

22. Создать класс *вектор на плоскости*, поля класса – координаты вектора. Методы класса: вычисление длины вектора, вывод характеристик вектора. В классе предусмотреть методы перегрузки операций: сложение, скалярное и векторное произведения векторов, вычисление угла между векторами.

23. Создать класс *квадратная матрица*, поля класса – размерность и элементы матрицы. Методы класса: проверка, является ли матрица симметричной ($A = A^T$), вывод матрицы. В классе предусмотреть методы перегрузки операций: сложение, вычитание, умножение матриц, добавление к матрице числа.

24. Создать класс *обыкновенная дробь*, поля класса – числитель и знаменатель. Метод класса: вывод дроби. В классе предусмотреть методы перегрузки операций: сложение, вычитание, деление и умножение дробей, сравнение дробей.

25. Создать класс *квадрат*, члены класса - длина стороны. Предусмотреть в классе методы вычисления и вывода сведений о фигуре – диагональ, периметр, площадь. Создать производный класс – *правильная квадратная призма с высотой H*, добавить в класс метод определения объема фигуры, перегрузить методы расчета площади и вывода сведений о фигуре.