

Создание графического приложения с использованием Windows Forms

1	Создание первого приложения	2
1.1	Установка VS2017 C++ дома	
1.2	Проверка установки Visual Studio	
1.3	Создание приложения Windows Forms	
1.4	Использование графических элементов в оконном приложении	
1.5	Задание для самостоятельной работы	
1.6	Контрольные вопросы	
1.7	Загрузка проекта на портал <i>course.sgu.ru</i>	
1.8	Некоторые замечания при работе с проектом	
2	Простое рисование на форме	21
2.1	Предварительная подготовка	
2.2	Определение рисунка	
2.3	Изменение соотношения сторон рисунка	
2.4	Задание для самостоятельной работы	
2.5	Контрольные вопросы	
3	Преобразования изображений	31
3.1	Предварительная подготовка	
3.2	Предмет реализации	
3.3	Геометрические преобразования	
3.4	Внутренний формат представления объектов	
3.5	Файловый ввод данных	
3.6	Очистка проекта	
3.7	Задание для самостоятельной работы	
3.8	Контрольные вопросы	

1. Создание первого приложения

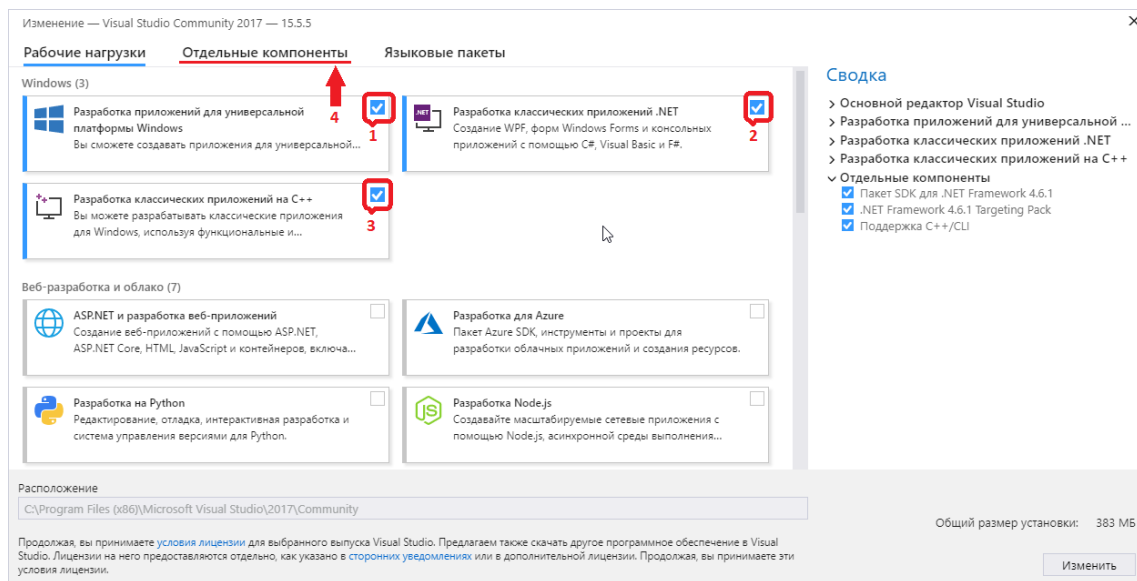
1.1 Установка VS2017 C++ дома

Первым делом вам необходимо посетить официальный сайт VS2017.

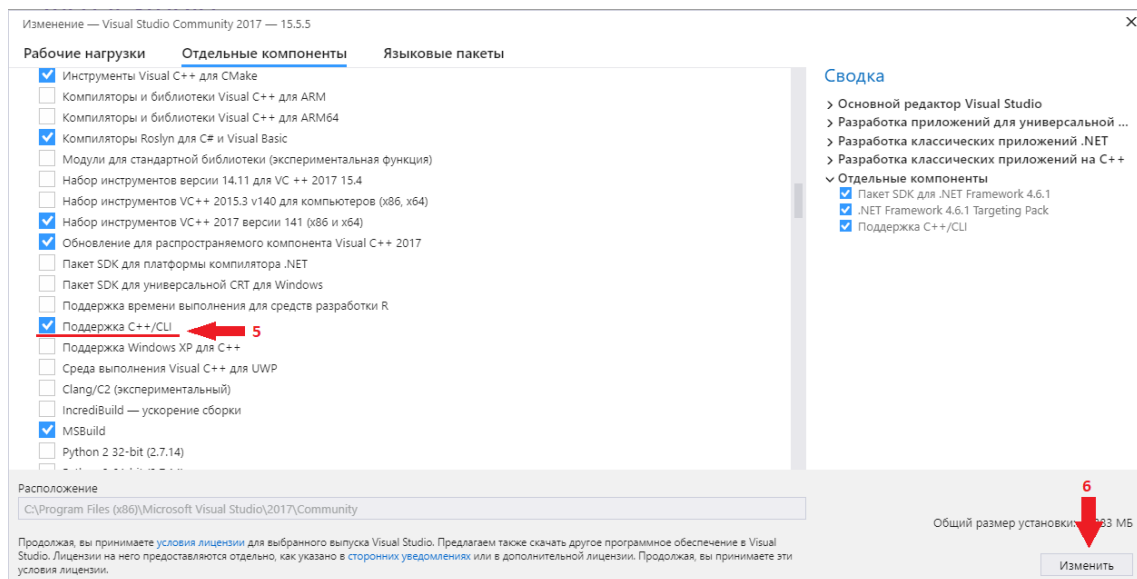


Выберите *Скачать Visual Studio* → *Community 2017*, после чего осуществится переход на страницу загрузки.

Запустите загруженный файл и переходите непосредственно к установке. Необходимый минимум компонентов отмечен пунктами 1–3 на рисунке.



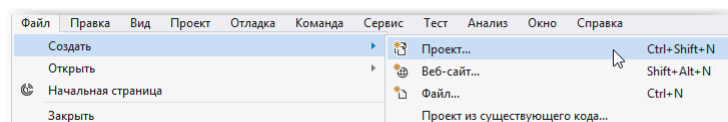
Помимо этого вам также требуется перейти на вкладку *Отдельные компоненты* (пункт 4) и отметить для установки пункт *Поддержка C++/CLI*.



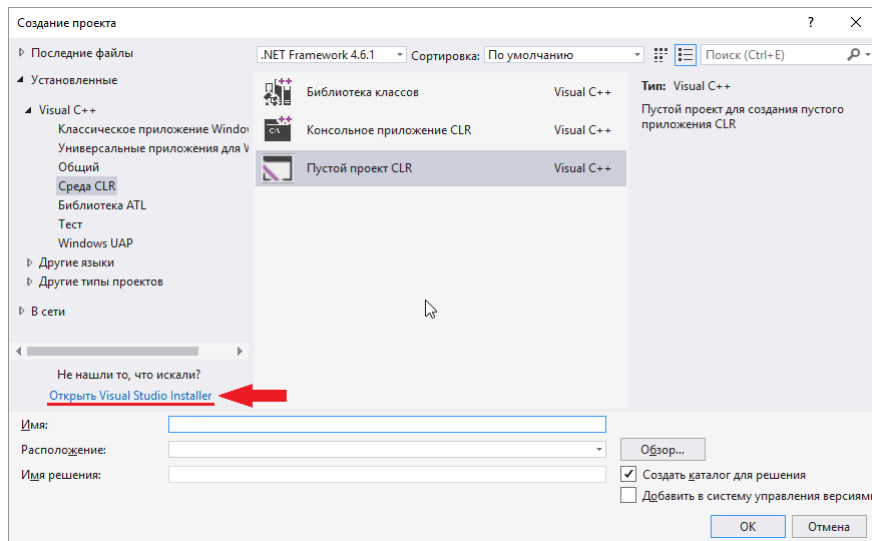
1.2 Проверка установки Visual Studio

Перед тем как создавать приложение Windows Forms необходимо сначала убедиться, что у Вас установлен компонент *Поддержка C++/CLI*.

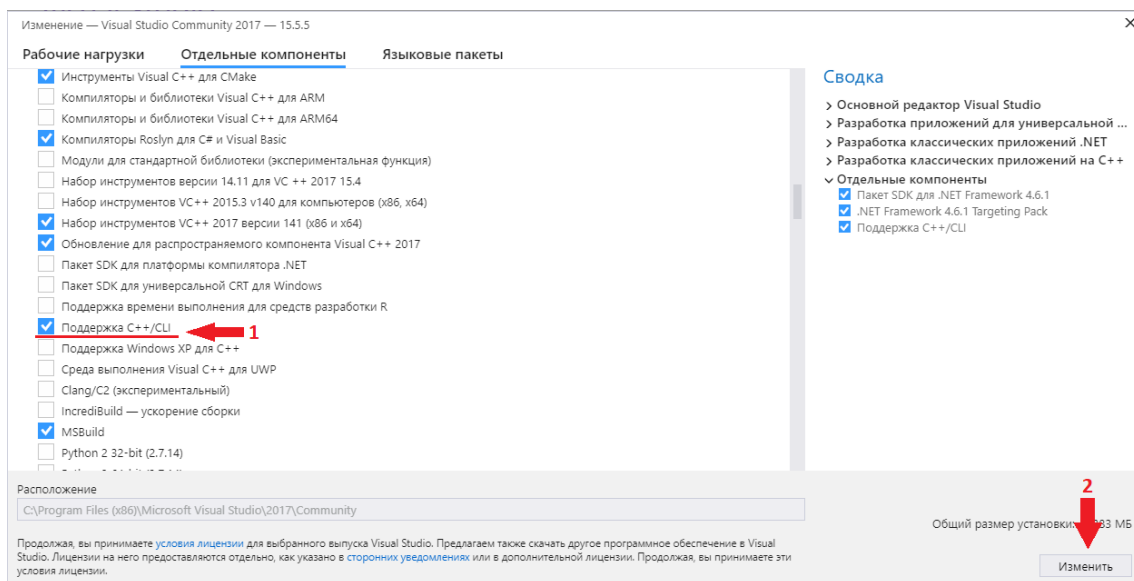
Для этого, после запуска Visual Studio, нужно выбрать пункт меню *Файл* → *Создать* → *Проект*.



В появившемся окне перейдите по ссылке *Открыть Visual Studio Installer*.



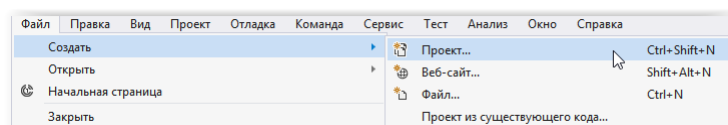
В окне установщика Visual Studio выберите вкладку *Отдельные компоненты* и проверьте, стоит ли галочка около пункта *Поддержка C++/CLI* (см. рисунок). Если галочки нет, то нужно её поставить и нажать кнопку *Изменить*.



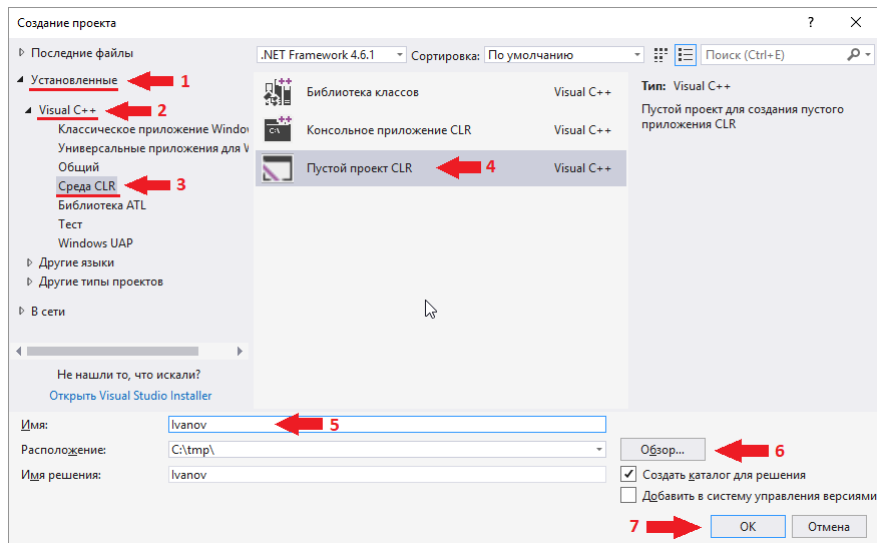
1.3 Создание приложения Windows Forms

1.3.1 Создание нового проекта

Для создания проекта Windows Forms следует выбрать пункт меню *Файл* → *Создать* → *Проект*.

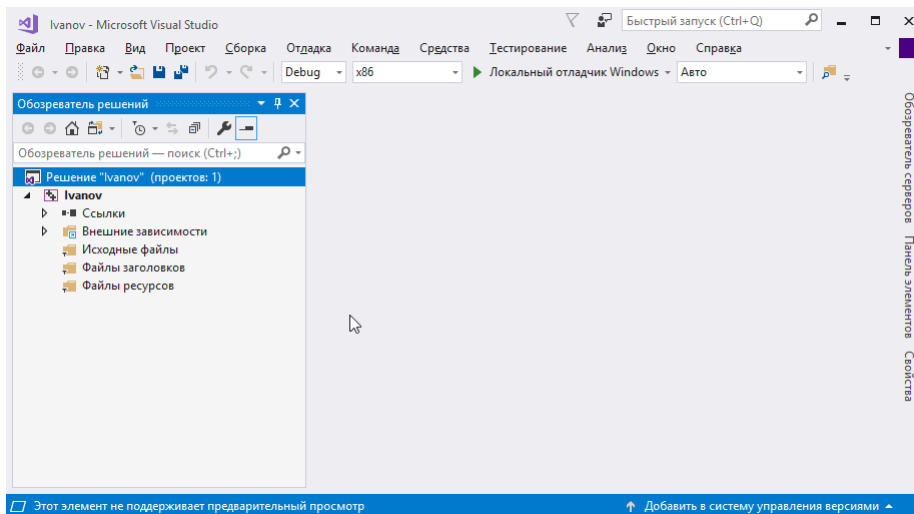


В появившемся окне нужно выбрать *Установленные* → *Visual C++* → *Среда CLR* → *Пустой проект CLR* (пункты 1–4 на рисунке).



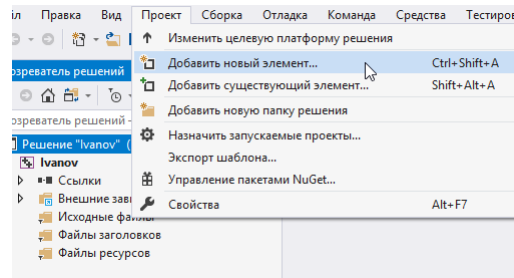
В нижней части этого же окна задайте имя создаваемого проекта — **Ваша фамилия латинскими буквами** (пункт 5). Обратите внимание на месторасположение будущего проекта (поле *Расположение*) и, если необходимо, поменяйте его (пункт 6). Для завершения создания проекта нажмите *OK* (пункт 7).

Visual Studio создаст пустой проект. Окно проекта будет выглядеть примерно так.

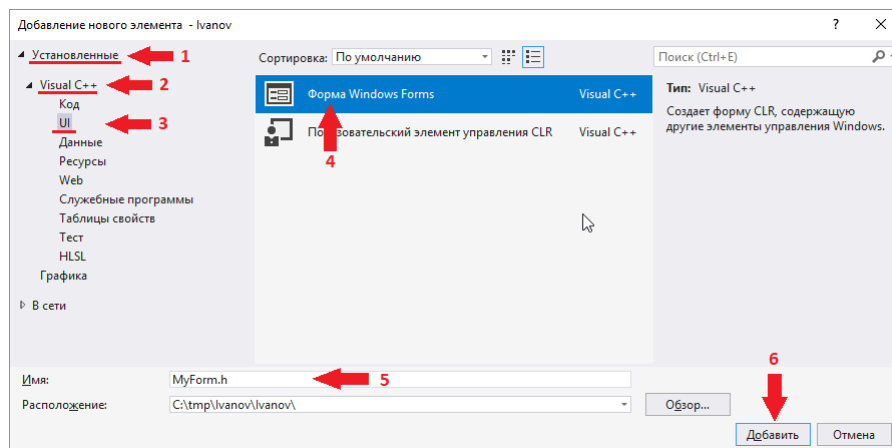


1.3.2 Добавление окна к проекту

Следующий шаг — добавление окна (формы) в приложение. Для этого необходимо выбрать пункт меню *Проект* → *Добавить новый элемент* (или нажать комбинацию клавиш **Ctrl + Shift + A**)



В появившемся окне нужно выбрать *Установленные* → *Visual C++* → *UI* → *Форма Windows Forms* (пункты 1–4 на рисунке).

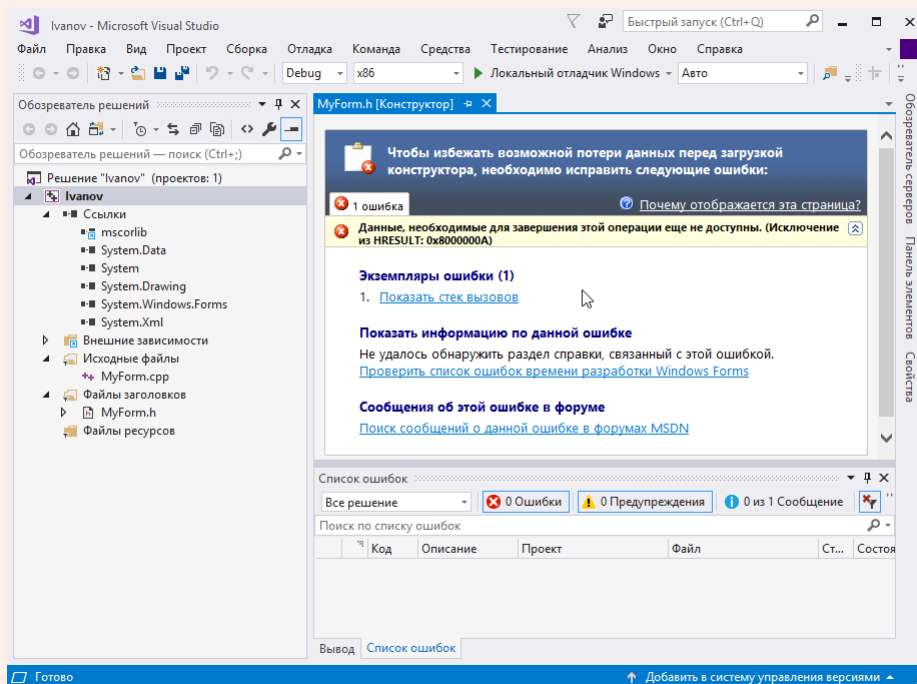


Название формы (пункт 5 на рисунке) можно поменять, но можно оставить `MyForm.h`. Для завершения создания формы нажмите *Добавить* (пункт 6).

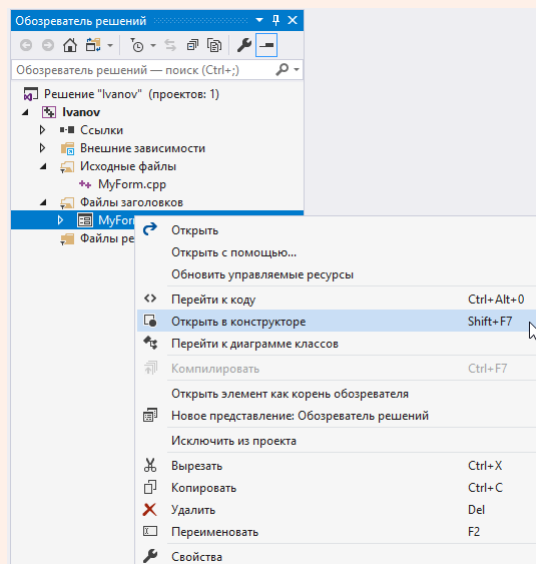


В дальнейшем здесь будем указывать имена, используемые по умолчанию. Обращайте внимание на те элементы приложения, которым Вы даёте собственное имя.

Очень часто при добавлении новой формы возникает ошибка

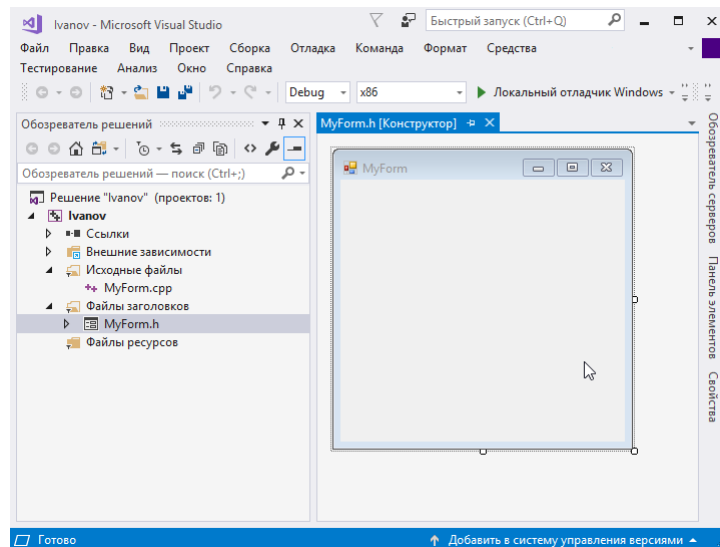


В случае, если ошибка возникла сразу после добавления окна, необходимо закрыть среду Visual studio и открыть заново. В открытом проекте закройте конструктор. В обозревателе решений нажмите правую кнопку мыши на имени файла с описанием формы (по умолчанию MyForm.h) и в контекстном меню выберите *Открыть в конструкторе*.

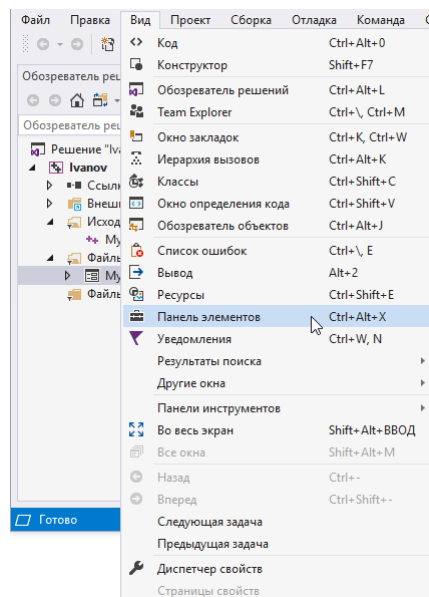


Если ошибка отображается после открытия проекта, то просто закройте конструктор и откройте его снова (как описано выше).

Если все сделано правильно, должна получиться форма, подобная представленной на рисунке.



К этой форме уже можно подключать различные элементы. Панель инструментов для размещения элементов на форме можно вывести выбрав пункт меню *Вид* → *Панель элементов*.



1.3.3 Функция Main

Чтобы созданный вами проект начал функционировать, требуется осуществить еще несколько действий.

Когда форма будет добавлена, в обозревателе решений выбираем файл `MyForm.cpp`. Перед вами откроется новая вкладка с единственной строкой кода: `#include "MyForm.h"`. Нужно в этом файле воспроизвести следующий код.



Здесь и далее постарайтесь не копировать код из методички, а писать его вручную: среда программирования поможет Вам с помощью системы автодополнения. Если Вы освоитесь с ней, то это позволит избежать ошибок в дальнейшем.


```

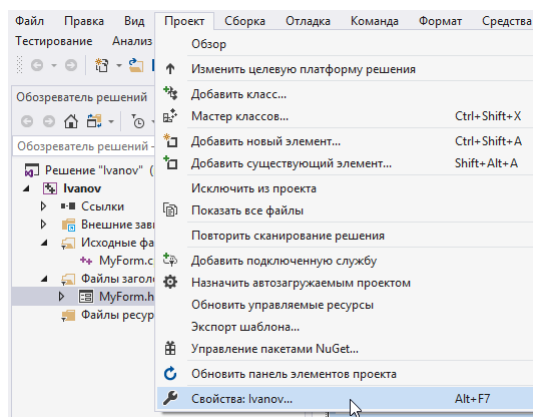
1  #include "MyForm.h"
2
3  using namespace System;
4  using namespace System::Windows::Forms;
5
6  [STAThreadAttribute]
7  void Main(array<String^>^ args) {
8      Application::EnableVisualStyles();
9      Application::SetCompatibleTextRenderingDefault(false);
10     Ivanov:MyForm form;
11     Application::Run(%form);
12 }

```

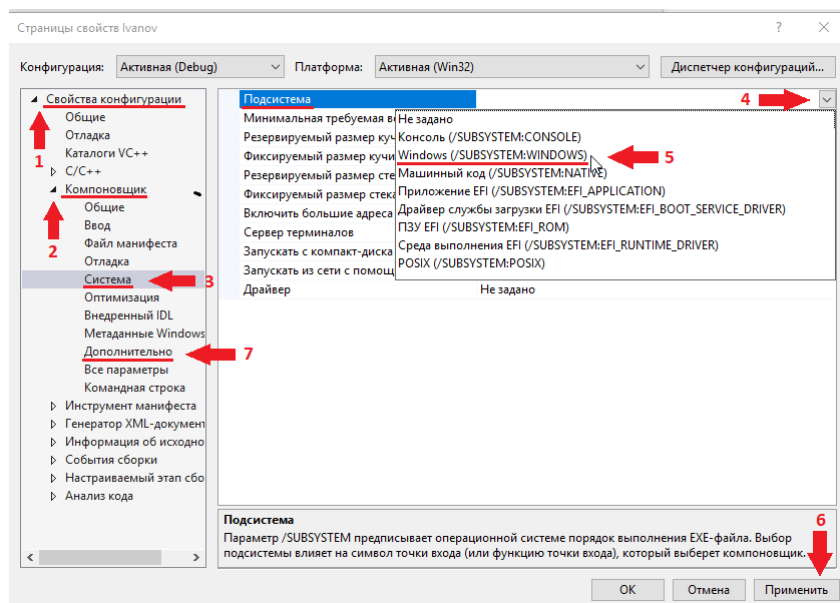
В строке 10 вместо имени Ivanov необходимо ввести **название Вашего проекта**. Сохранить изменения.

1.3.4 Установка свойств проекта

На панели *Обозреватель решений* нажмите правой кнопкой мыши на название Вашего проекта и выберите *Свойства* или выберите пункт меню *Проект → Свойства*.

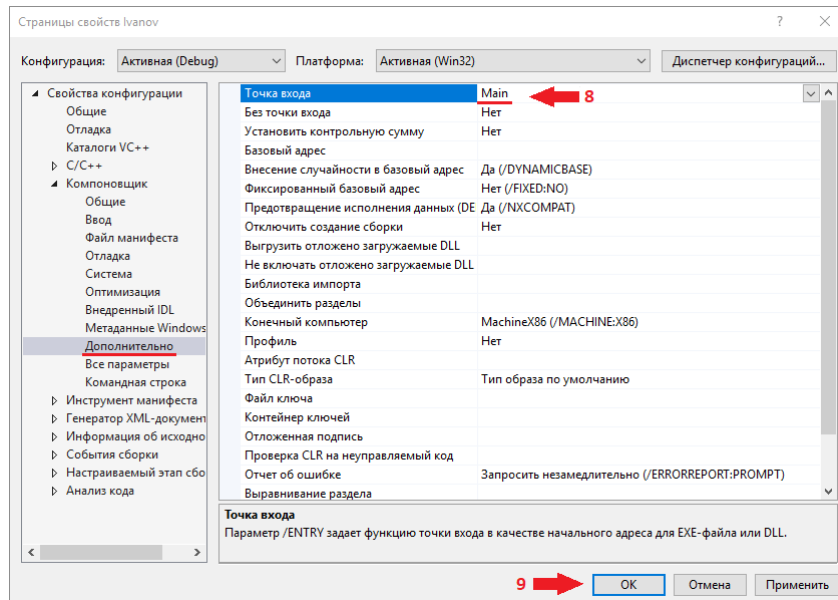


В появившемся окне выберите *Свойства конфигурации* → *Компоновщик* → *Система* (пункты 1–3 на рисунке).



В строке *Подсистема* выберите Windows (/SUBSYSTEM:WINDOWS) (пункты 4–5) и нажмите кнопку *Применить* (пункт 6). В этом же окне перейдите к разделу *Дополнительно* (пункт 7).

На открывшейся странице, в строке *Точка входа* следует набрать Main (пункт 8) и нажать кнопку *OK* (пункт 9).

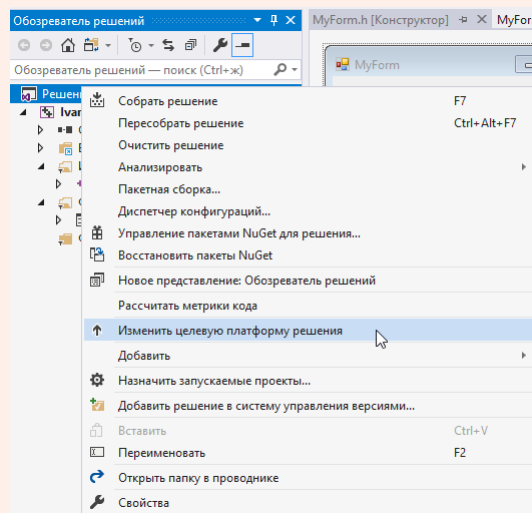


Теперь проект можно запустить.

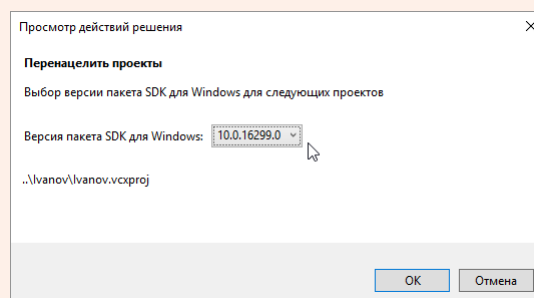
Частой ошибкой компиляции приложения Visual Studio 2017 является ошибка, связанная с неправильно указанной версией SDK в свойствах проекта. Например

```
1>C:\Program Files (x86)\Microsoft Visual
Studio\2017\Community\Common7\IDE\VC\VCTargets\Platforms\Win32\PlatformToolsets
\v141\Toolset.targets(34,5): error MSB8036: не удалось найти Windows SDK версии
8.1. Установите нужную версию Windows SDK или измените версию SDK на страницах
свойств проекта либо щелкнув правой кнопкой мыши решение и выбрав "Изменить
целевую платформу решения".
```

Чтобы исправить ошибку, можно следовать инструкции, указанной в данном сообщении: в обозревателе решений кликнуть правой кнопкой мыши на заголовке решения и в контекстном меню выбрать *Изменить целевую платформу решения*.



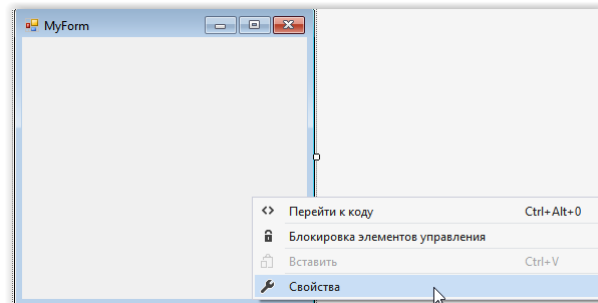
В появившемся окне из выпадающего списка выберите одну из предложенных версий SDK.



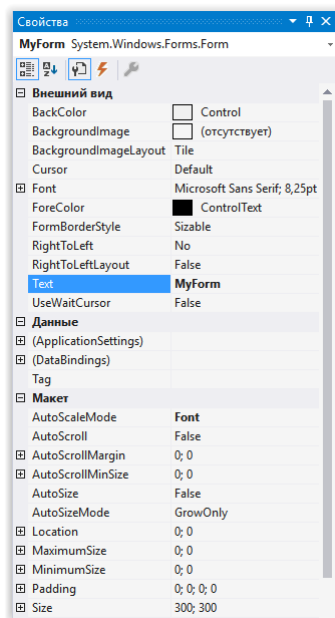
1.4 Использование графических элементов в оконном приложении

1.4.1 Обработчик события Paint

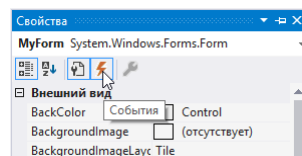
Нажмите правую кнопку мыши на форме и в появившемся меню выберите пункт *Свойства*.



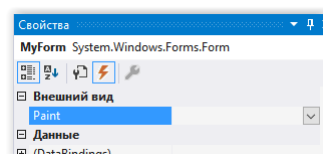
Появится окно диспетчера свойств, в котором отражены все параметры созданной формы.



Перейдите на вкладку *События* диспетчера свойств (см. рисунок).



Среди перечисленных событий найдите событие *Paint*.



Обработчик события *Paint* ответственен за графическое наполнение формы. Конечно же, событие *Paint* возникает сразу при запуске формы. Кроме того, это событие возникает когда Вы разворачиваете окно, после того, как предварительно свернули его на панель задач.

Если мы собираемся что-то рисовать на форме, то это что-то должно перерисовываться каждый раз, когда возникает событие *Paint*. Поэтому, обработчик этого события есть подходящее место для команд, графического вывода. В качестве пробного шага закрасим форму цветом и выведем на ней несколько линий и надпись.

Для того, чтобы добавить обработчик события, дважды кликните мышью на пустом поле правее имени *Paint*. В файле, описывающем форму (по умолчанию *MyForm.h*), будет создана процедура — обработчик события отрисовки формы и вы будете перенаправлены для редактирования этой процедуры.

```
1 private: System::Void MyForm_Paint(System::Object^ sender, System::Windows::Forms::PaintEventArgs^ e) {  
2 }
```

Сразу нажмем на клавишу *Enter* (чтобы вставить пустую строку) и приступим к изменению данной процедуры.

Все команды рисования на форме обычно имеют вид

область_рисования->команда(аргументы)

где область рисования — ссылка на объект типа *System::Drawing::Graphics^*. Такую ссылку для нашего окна приложения можно извлечь из аргумента *e* процедуры *MyForm_Paint* (аргумент типа *System::Windows::Forms::PaintEventArgs^*), обращаясь к свойству *Graphics*: *e->Graphics*.

Так как в нашей программе будет производиться много обращений к области рисования, опишем переменную *g*, в которой сохраним это значение.

```
System::Drawing::Graphics^ g = e->Graphics;
```

Так как пространство имен *System::Drawing* уже подключено (в начале файла *MyForm.h*), достаточно описать:

```
Graphics^ g = e->Graphics;
```

Теперь переменная *g* ссылается на область рисования в нашей форме.

1.4.2 Закраска области рисования

Для того, чтобы закрасить всю область для рисования, следует воспользоваться методом очистки области *Clear*, которой в качестве аргумента передается цвет (значение типа *System::Drawing::Color*). Очистка области состоит в том, что метод заливает всю область заданным цветом.



Если в редакторе Visual Studio набрать *System::Drawing::Color::*, то система автодополнения предложит Вам список всех цветов, к которым можно обратиться по имени.

Помня, что часть *System::Drawing::* можно опустить, закрасим (очистим) область *g* цветом «Аквамарин». Для этого дадим команду:

```
g->Clear(Color::Aquamarine);
```



Здесь и далее при написании кода часто будем опускать префиксы подключенных пространств имен. Однако часто, в случаях, когда вы забыли название типа/метода/свойства, удобно писать забытое имя начиная с префикса пространства имен, чтобы система автодополнения подсказала Вам возможные названия.

1.4.3 Вычерчивание линий

Сначала начертим толстую красную линию, соединяющую верхний левый и правый нижний угол.

Для черчения следует определить перо — элемент типа `System::Drawing::Pen`. У пера должны быть определены две характеристики: цвет (`Color`) и толщина (`Width`). В конструкторах для пера толщина может быть принята как значение по умолчанию, в то время как цвет должен быть явно (или косвенно) указан. Создадим перо `redPen` красного цвета, воспользовавшись первым предложенным системой конструктором, после чего изменим толщину пера на значение 6:

```
Pen^ redPen = gcnew Pen(Color::Red);
redPen->Width = 6;
```



Можно было воспользоваться конструктором, в котором указывается и цвет, и толщина. Приведенный пример лишь демонстрирует, что свойства пера можно изменить по ходу работы программы. Аналогичным образом, у пера можно установить и новое значение цвета.

Для того, чтобы начертить отрезок, нужно знать координаты начала и конца отрезка. Система координат в окне — левосторонняя. Начало координат находится в верхнем левом углу окна, так что с точкой начала отрезка все ясно.

Чтобы узнать координаты правого нижнего угла, следует обратиться к свойствам формы, извлечь ширину и высоту области рисования.

Так как мы описываем метод класса `MyForm`, можем воспользоваться методами родительского класса `System::Windows::Forms::Form`, извлекая их имена с помощью префикса `this->` (вместо этого префикса можно использовать полный префикс `System::Windows::Forms::Form::` или просто `Form::`, т.к. пространство имен `System::Windows::Forms` уже подключено).

Казалось бы логичным, что координаты нижнего правого угла должны быть с координатой x равной `this->Width` и с координатой y равной `this->Height`. Но такое предположение является ошибкой, так как `this->Width` и `this->Height` — ширина и высота всего окна, включая заголовок и рамки (при их наличии). Нам же нужны высота и ширина внутренней части окна. Их извлечь нам поможет значение свойства `ClientRectangle` — прямоугольник (элемент типа `System::Drawing::Rectangle`), занимающий всю видимую часть области рисования.

Используя вышесказанное, начертим пером `redPen` линию

```
g->DrawLine(redPen, 0, 0, this->ClientRectangle.Width, this->ClientRectangle.Height);
```



Префикс `this->` может быть полезным в смысле общения с системой автодополнения, в частности, чтобы узнать, какие свойства и методы уже определены у нашего окна. Но так как он отсылает нас к текущему (активному) пространству имен, то его наличие факультативно.

Начертим синим пером линию еще большей толщины от координат (90,50) до точки, касающейся правой стороны окна, на высоте 80.

```
Pen^ bluePen = gcnew Pen(Color::Blue, 10);
g->DrawLine(bluePen, 90, 50, ClientRectangle.Width, 80);
```

1.4.4 Вывод текста

Текст выводится кистью (элемент типа `System::Drawing::SolidBrush^`). Кисть, как и перо, имеет атрибут — цвет, который указывается при инициализации. Опишем переменную `drawBrush` — экземпляр кисти черного цвета.

```
SolidBrush^ drawBrush = gcnew SolidBrush(Color::Black);
```

Кроме того, для письма необходим шрифт — элемент типа `System::Drawing::Font^`, характеристиками которого являются имя и размер.

Опишем переменную `drawFont` — экземпляр шрифта Arial размером 10.

```
System::Drawing::Font^ drawFont = gcnew System::Drawing::Font("Arial", 10);
```



Если опустить префикс `System::Drawing::` в последнем фрагменте, то система будет выдавать ошибку при компиляции. Чтобы этого избежать, следует использовать перед `Font` префикс `System::Drawing::` или `Drawing::`.

Выведем надпись «Надпись на форме» по координатам (40, 100) созданной ранее кистью черного цвета и шрифтом Arial размером 10.

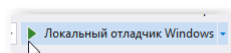
```
g->DrawString("Надпись на форме", drawFont, drawBrush, 40, 100);
```

На этом завершим редактирование процедуры. После редактирования она будет выглядеть так:

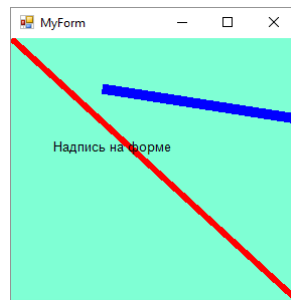
```
1 private: System::Void MyForm_Paint(System::Object^ sender, System::Windows::Forms::PaintEventArgs^ e) {
2     Graphics^ g = e->Graphics;
3     g->Clear(Color::AquaMarine);
4
5     Pen^ redPen = gcnew Pen(Color::Red);
6     redPen->Width = 6;
7     g->DrawLine(redPen, 0, 0, this->ClientRectangle.Width, this->ClientRectangle.Height);
8
9     Pen^ bluePen = gcnew Pen(Color::Blue, 10);
10    g->DrawLine(bluePen, 90, 50, ClientRectangle.Width, 80);
11
12    SolidBrush^ drawBrush = gcnew SolidBrush(Color::Black);
13    System::Drawing::Font^ drawFont = gcnew System::Drawing::Font("Arial", 10);
14    g->DrawString("Надпись на форме", drawFont, drawBrush, 40, 100);
15 }
```

1.4.5 Компиляция и запуск

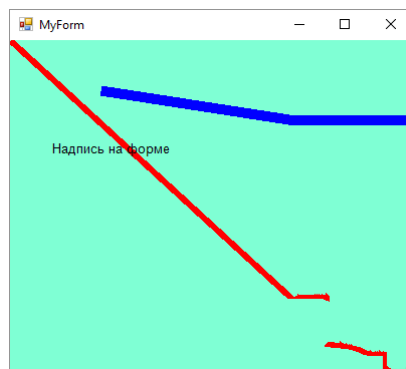
Полученное приложение можно скомпилировать. Для этого можно на панели инструментов нажать соответствующую кнопку.



После компиляции приложение будет запущено:

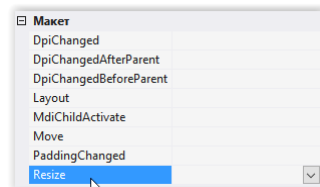


Все отрисовывается именно так, как мы планировали. Но при изменении размера формы рисунок не обновляется, а на форме появляется разный мусор.



Дело в том, что событие `Paint` не возникает при изменении размера формы. Чтобы это исправить, можно самостоятельно инициировать событие `Paint` с помощью вызова метода `this->Refresh()` после изменения размера формы.

Сначала закроем запущенное приложение, после чего в окне свойств формы во вкладке *События* найдем обработчик события *Resize*.



Дважды кликните мышкой на пустом поле справа от наименования *Resize*. Будет создана процедура — обработчик события, срабатывающая при изменении размера формы.

```
1 private: System::Void MyForm_Resize(System::Object^ sender, System::EventArgs^ e) {  
2 }
```

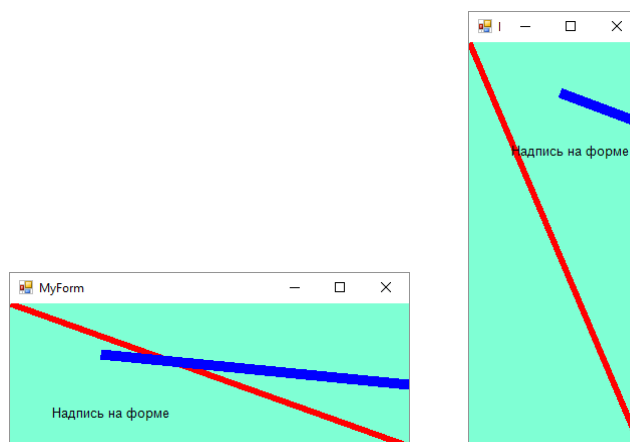
В теле этой процедуры дадим всего лишь одну команду:

```
Refresh();
```

После редактирования процедура будет выглядеть так:

```
1 private: System::Void MyForm_Resize(System::Object^ sender, System::EventArgs^ e) {  
2     Refresh();  
3 }
```


После повторной компиляции и запуска изменение размера формы приведет к полному обновлению рисунка на форме.

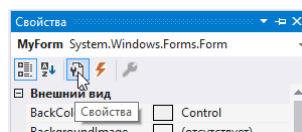


1.4.6 Двойная буферизация

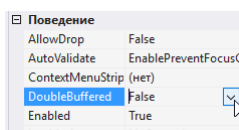
До сих пор мы рисовали непосредственно на форме. Такой принцип рисования плох тем, что при частом обновлении рисунка на форме (например, при наличии динамики в изображении) изображение может мерцать: форма сначала очищается, а затем элемент за элементом происходит отрисовка. Кроме того, побочным эффектом является низкая скорость отрисовки.

Выходом из этой ситуации является включение двойной буферизации. В этом случае изображению в окне будет соответствовать два участка памяти (буфера): один содержит отображаемое в текущий момент изображение (первичный буфер), а второй — для отрисовки следующего «кадра» (вторичный буфер). Команды рисования будут приводить к изменению не первичного, а вторичного буфера. Только после того, как обработчик события `Paint` закончит свою работу, содержимое первичного буфера заменится на содержимое вторичного буфера. Так как непосредственно в окне будет выполнена лишь одна графическая операция (полностью заменено одно изображение на другое), то мерцание исчезнет и, зачастую, заметно повысится скорость работы.

По умолчанию, режим двойной буферизации отключен. Для того, чтобы его включить в панели свойств окна переключитесь на страницу *Свойства*.



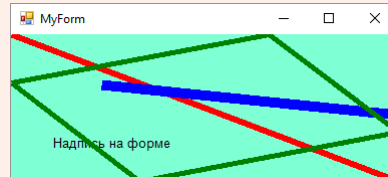
Среди перечисленных свойств найдите `DoubleBuffered` и измените его значение на `True`.



1.5 Задание для самостоятельной работы

Задание 1

1. Создайте приложение, включающее в себя все, что было описано выше в качестве примера. Проект должен называться Вашей фамилией, записанной латинскими буквами.
2. Добавьте в получившийся проект следующее дополнение. Определите перо зеленого цвета толщины 5. С помощью этого пера соедините точки на сторонах формы, делящие каждую сторону в соотношении 2:1 так, как показано на рисунке.



3. Архив получившегося проекта загрузите на портал как ответ на задание 1 (см. раздел 1.7 о составе архива).

1.6 Контрольные вопросы

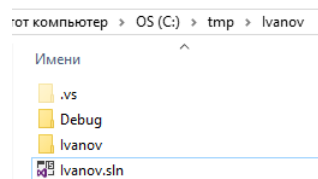
Подготовьте ответы на следующие вопросы.

1. Как определить размер формы по горизонтали и вертикали?
2. Как две точки на форме соединить отрезком?
3. Как установить цвет и толщину линии отрезка?
4. Как закрасить фон на форме?
5. Как вывести строку в определенных координатах на форме?
6. Как настроить атрибуты шрифта выводимой строки?
7. Как перерисовать (обновить) изображение на форме?
8. В какой момент запускается обработчик события *Paint*?
9. Как привязать конец отрезка к точке, отстоящей от правого нижнего угла окна на заданное расстояние по горизонтали и вертикали?

1.7 Загрузка проекта на портал course.sgu.ru

Так как размер загружаемого файла может быть не более 1 Мб, необходимо загружать архив, содержащий только нужные компоненты.

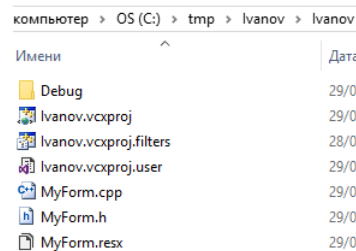
Например, при создании проекта `Ivanov`, в папке с проектом (её первоначальное название, обычно, такое же как и у проекта) создаются файлы и папки, показанные на рисунке.



Папка `.vs` обычно является скрытой. При некоторых предварительных установках папка может еще содержать папку `Release`.

В ваш архив с решением необходимо отсюда поместить только файл с названием проекта и типом *Microsoft Visual Studio Solution* (имеющий расширение `.sln`, но расширения у Вас могут не отображаться), а так же папку с тем же именем.

Но прежде чем переходить к архивации зайдите в папку с именем проекта. Содержимое этой папки может быть следующим.



компьютер > OS (C:) > tmp > Ivanov > Ivanov	
Имени	Дата
Debug	29/0'
Ivanov.vcxproj	29/0'
Ivanov.vcxproj.filters	28/0'
Ivanov.vcxproj.user	29/0'
MyForm.cpp	29/0'
MyForm.h	29/0'
MyForm.resx	29/0'

Удалите здесь папки `Debug` и `Release` (если они там присутствуют).

После удаления вернитесь в предыдущую папку и поместите нужные файл и папку в архив. Имя архива должно быть `task_1_Ivanov`, где вместо слова `Ivanov` должна стоять Ваша фамилия, записанная латиницей.

Для последующих заданий в имени архива должен изменяться номер задания.

Полученный архив нужно загрузить в качестве решения.

1.8 Некоторые замечания при работе с проектом

Для того, чтобы перейти из конструктора формы к программному коду, можно использовать три способа:

1. Нажать правой кнопкой мыши по любому месту конструктора и в появившемся окне выбрать *Перейти к коду* (или нажать `F7`).
2. В панели свойств окна, на странице *События* дважды кликнуть левой кнопкой мыши на поле, соответствующем нужному событию.
3. Дважды кликнуть левой кнопкой мыши на нужном элементе формы (кнопка, текстовое поле и т. д.).



Если Вы не хотите, чтобы создавались какие-то методы, а лишь желаете перейти к редактированию уже существующего кода, то первый способ является самым предпочтительным. Последний способ следует применять только если Вы точно осознаете свои действия.

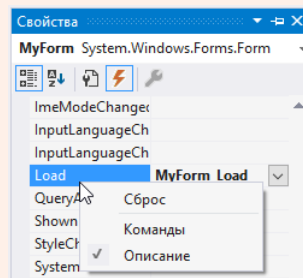
После любого из перечисленных выше действий Вы перейдете к редактированию программного кода, соответствующего окну.

Если Вы дважды кликните левой кнопкой мыши по форме в конструкторе или по ее элементу, то автоматически будет создана (если еще не создавалась) какая-то процедура — обработчик события (тип обработчика зависит от элемента, по которому Вы кликнули). Например, если Вы кликните на пространстве формы, то будет создан обработчик события `Load`.

Если это произошло, но Высшей целью не было создание такой процедуры, не следует сразу удалять её текст из файла.

Сначала вернитесь в конструктор формы и откройте панель свойств того элемента для которого был создан обработчик события. На вкладке *События* кликните правой

кнопкой мыши на имени соответствующего события (справа от него будет вписано имя созданной процедуры) и из выпадающего меню выберите пункт *Сброс*.



Тем самым мы отвязали созданную процедуру от окна. Если Вы не вносили своих изменений в эту процедуру, то система автоматически удалит ее из кода формы. Если изменения все же вносились, и Вы хотите удалить эту процедуру из проекта, то теперь можно перейти к редактированию кода (желательно первым способом) и удалить текст процедуры из файла.



В некоторых случаях одна и та же процедура может являться обработчиком разных событий. Назначить уже существующую процедуру обработчиком некоторого события можно на вкладке *События* панели свойств. Важно, при удалении процедуры из файла с программным кодом, предварительно сбросить все привязки этой процедуры к событиям.



Необходимо следить, чтобы случайно не стереть нужные элементы кода. Файл ОБЯЗАТЕЛЬНО должен заканчиваться двумя закрывающими скобками: первая закрывающая скобка с точкой с запятой «`};`» закрывает класс `MyForm`, а вторая закрывает `namespace Ivanov` — пространство имен, совпадающее с названием Вашего проекта (поэтому и следует называть проект латинскими буквами).

2. Простое рисование на форме

2.1 Предварительная подготовка

За основу проекта для второго задания возьмем уже готовый проект, получившийся в результате выполнения первого задания.

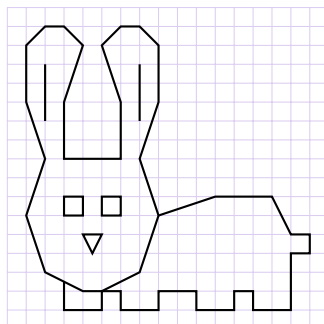
Сделайте копию папки с проектом первого задания (чтобы не повторять предварительные установки). Имя проекта должно остаться без изменений.

Перед Вами ставится задача изобразить рисунок, соответствующий Вашему варианту (следует предварительно выбрать номер свободного варианта на портале course.sgu.ru: ссылка «Выбор варианта для задания 2»).

В рамках этого задания Вы, главным образом, будете изменять обработчик события *Paint* формы. Следует предварительно убрать из этой процедуры инструкции, выводящие линии и текст на форму. Цвет фона остается на Ваше усмотрение, так же как цвет и толщина линий, которыми Вы будете выполнять собственное задание.

2.2 Определение рисунка

В качестве задания Вашего варианта выступает рисунок, исполненный в некоторой координатной сетке. Здесь в качестве примера возьмём следующее изображение.



Клетки на рисунке даны лишь для привязки концов отрезка к некоторым координатам и для определения пропорций рисунка.

Ваша задача — вывести все отрезки, составляющие данный рисунок.

Попробуем нарисовать линии, из которых состоит данный рисунок. Сначала свяжем с рисунком систему координат. Пусть начало координат находится в левом нижнем углу, а две клетки соответствуют одной единице. Тогда можно перечислить все отрезки из которых состоит рисунок.

Переместитесь в самое начало файла `MyForm.h`. Здесь, после объявления пространства имен, можно видеть несколько команд `using namespace`. После этих команд (до начала описания класса `MyForm`) будем добавлять описание глобальных переменных.

Сейчас здесь опишем глобальный массив вещественных чисел, в который сохраним все отрезки: каждая четверка элементов массива будет соответствовать двум парам координат концов очередного отрезка.

```
float lines[] = {
    // голова
    0.5f,3.f,1.f,4.5f, // от левой щеки вверх до уха
    1.f,4.5f,0.5f,6.f, // левое ухо слева снизу вверх
    0.5f,6.f,0.5f, 7.5f, // левое ухо слева
    0.5f, 7.5f,1.f,8.f, // левое ухо верх слева
    1.f,8.f,1.5f,8.f, // левое ухо верх середина
    1.5f,8.f,2.f,7.5f, // левое ухо верх справа
    2.f,7.5f,1.5f, 6.f, // левое ухо справа сверху вниз
    1.5f, 6.f,1.5f,4.5f, // левое ухо справа до макушки
    1.5f,4.5f,3.f,4.5f, // макушка
    3.f,4.5f,3.f,6.f, // правое ухо слева снизу вверх
    3.f,6.f,2.5f,7.5f, // правое ухо слева
    2.5f,7.5f,3.f,8.f, // правое ухо верх слева
    3.f,8.f,3.5f,8.f, // правое ухо верх середина
    3.5f,8.f,4.f,7.5f, // правое ухо верх справа
    4.f,7.5f,4.f,6.f, // правое ухо сверху вниз
    4.f,6.f,3.5f,4.5f, // правое ухо справа
    3.5f,4.5f,4.f,3.f, // от правого уха вниз до щеки
    4.f,3.f,3.5f,1.5f, // правая скула
    3.5f,1.5f,2.5f,1.f, // подбородок справа
    2.5f,1.f,2.f,1.f, // подбородок снизу
    2.f,1.f,1.f,1.5f, // подбородок слева
    1.f,1.5f,0.5f,3.f, // левая скула
    // туловище
    4.f,3.f,5.5f,3.5f, // спина от головы вправо
    5.5f,3.5f,7.f,3.5f, // спина вверх
    7.f,3.5f,7.5f,2.5f, // спина сверху до хвоста
    7.5f,2.5f,8.f,2.5f, // хвост сверху
    8.f,2.5f,8.f,2.f, // хвост справа
    8.f,2.f,7.5f,2.f, // хвост низ справа налево
    7.5f,2.f,7.5f,0.5f, // задняя нога справа сверху вниз
    7.5f,0.5f,6.5f,0.5f, // задняя нога низ
    6.5f,0.5f,6.5f,1.f, // задняя нога слева
    6.5f,1.f,6.f,1.f, // между задних ног
    6.f,1.f,6.f,0.5f, // левая задняя нога справа
    6.f,0.5f,5.f,0.5f, // левая задняя нога низ
    5.f,0.5f,5.f,1.f, // левая задняя нога слева
    5.f,1.f,4.f,1.f, // между задними и передними ногами
    4.f,1.f,4.f,0.5f, // правая передняя нога справа
    4.f,0.5f,3.f,0.5f, // правая передняя нога низ
    3.f,0.5f,3.f,1.f, // правая передняя нога слева
    3.f,1.f,2.5f,1.f, // между передних ног
    2.5f,1.f,2.5f,0.5f, // передняя нога справа
    2.5f,0.5f,1.5f,0.5f, // передняя нога низ
    1.5f,0.5f,1.5f,1.25f, // передняя нога слева
}
```

```
// левый глаз
1.5f,3.5f,1.5f,3.f, // левый глаз слева сверху вниз
1.5f,3.f,2.f,3.f, // левый глаз низ
2.f, 3.f,2.f,3.5f, // левый глаз справа
2.f,3.5f,1.5f,3.5f, // левый глаз верх
// правый глаз
2.5f,3.5f,2.5f,3.f, // правый глаз слева
2.5f,3.f,3.f, 3.f, // правый глаз снизу
3.f,3.f,3.f,3.5f, // правый глаз справа
3.f,3.5f,2.5f,3.5f, // правый глаз сверху
// ушные раковины
1.f,5.5f,1.f,7.f, // левая ушная раковина
3.5f,5.5f,3.5f,7.f, // правая ушная раковина
// нос
2.f,2.5f,2.5f,2.5f, // нос сверху
2.5f,2.5f,2.25f,2.f, // нос справа
2.25f,2.f,2.f,2.5f // нос слева
};
```

Здесь же добавим целочисленную глобальную переменную, в которой сохраним количество отрезков.

```
unsigned int arrayLength = sizeof(lines) / sizeof(float);
```

Теперь вернемся к редактированию обработчика события *Paint*.

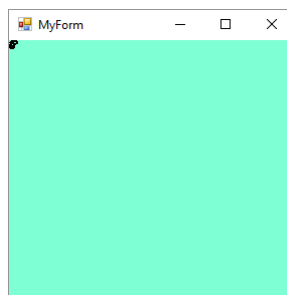
Нужно вывести каждый отрезок на форму. Для этого сначала создадим перо для вычерчивания

```
Pen^ blackPen = gcnew Pen(Color::Black, 2);
```

Теперь в цикле выведем все отрезки

```
for (int i = 0; i < arrayLength; i += 4) {
    g->DrawLine(blackPen, lines[i], lines[i + 1], lines[i + 2], lines[i + 3]);
}
```

Но после запуска мы увидим следующую картину

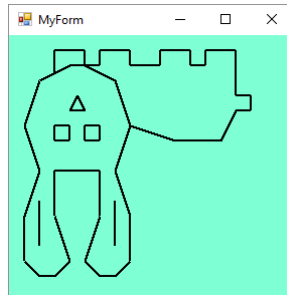


Черные вкрапления в верхнем левом углу как раз и есть наше изображение. Так как мы выводили отрезки без масштабирования координат, максимальная координата точки не превышает 8.5, а следовательно размер нашей картинки на экране не превышает девяти пикселей как в ширину, так и в высоту.

Чтобы наше изображение приняло приемлемый размер, определим перед циклом коэффициент масштабирования, на который домножим каждую из координат.


```
float S = 30.f; // коэффициент увеличения
for (int i = 0; i < arrayLength; i += 4) {
    g->DrawLine(blackPen, S * lines[i], S * lines[i + 1]
                , S * lines[i + 2], S * lines[i + 3]);
}
```

Теперь изображение примет следующий вид.

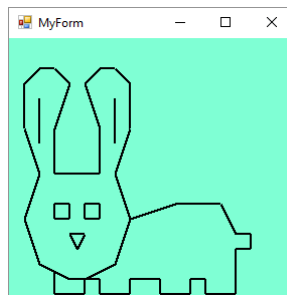


Размер изображения вполне приемлем. Но так как ось Oy в окне направлена сверху вниз, то картинка остается перевернутой относительно горизонтальной оси.

Для того, чтобы рисунок принял нужное положение, умножим (при выводе отрезка) каждую координату y на -1 . В этом случае все изображение перевернется, но пропадет из поля зрения, так как оно полностью переместится в отрицательную часть оси Oy (так как начало координат в верхнем левом углу). Чтобы сместить его в видимую часть окна, добавим к каждой получившейся координате y одну и ту же положительную величину Ty . Например:

```
float S = 30.f; // коэффициент увеличения
float Ty = 270.f; // смещение в положительную сторону по оси Oy после смены знака
for (int i = 0; i < arrayLength; i += 4) {
    g->DrawLine(blackPen, S * lines[i], Ty - S * lines[i + 1]
                , S * lines[i + 2], Ty - S * lines[i + 3]);
}
```

Изображение примет следующий вид.



Но величины S и Ty сейчас ничем не обусловлены и им присвоены «магические» числа, которые непонятно откуда взялись. Чтобы придать этим величинам смысл, давайте условимся, что

1. какой бы ни был размер окна, заданное изображение должно быть максимально возможно увеличено так, чтобы полностью помещаться в окне;
2. точка левого верхнего угла левой верхней клетки заданного изображения должна совпадать с левой верхней угловой точкой.

Для этого нам понадобятся величины V_x и V_y — размеры заданного рисунка в единицах нашей выбранной системы координат, а так же понадобятся размеры видимой части области рисования W_x и W_y .

Значения V_x и V_y присущи рисунку и не зависят от формы, поэтому опишем их как глобальные (сразу после описания массива `arrayLength`).

```
float Vx = 8.5f; // размер рисунка по горизонтали
float Vy = 8.5f; // размер рисунка по вертикали
```

Значения W_x и W_y вычислим в обработчике *Paint*.

```
float Wx = ClientRectangle.Width; // размер окна по горизонтали
float Wy = ClientRectangle.Height; // размер окна по вертикали
```

Кроме того, вычислим значения `aspectFig` и `aspectForm` — соотношения сторон нашего рисунка и окна. Как и в предыдущем случае `aspectFig` опишем как глобальную переменную.

```
float aspectFig = Vx / Vy; // соотношение сторон рисунка
```

Переменная `aspectForm` — локальная для обработчика события *Paint*.

```
float aspectForm = Wx / Wy; // соотношение сторон окна рисования
```

Зная эти величины, мы можем определить, каков должен быть коэффициент увеличения S , чтобы рисунок полностью поместился в окне: если значение соотношения сторон рисунка меньше чем соотношение сторон окна, то следует увеличивать в W_y / V_y раз, а в противном случае в W_x / V_x раз.

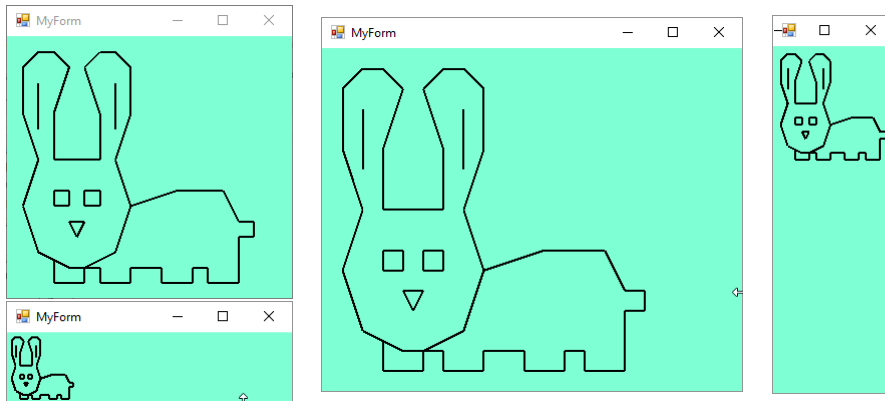
```
float S = aspectFig < aspectForm ? Wy / Vy : Wx / Vx; // коэффициент увеличения
```

Для того, чтобы выполнилось и второе условие, нужно определить правильную величину смещения рисунка по оси Oy .

Если рисунок не смещать, то точка начала координат рисунка совпадает с точкой начала координат окна. Нам же нужно совместить с точкой начала координат окна верхний левый угол увеличенного рисунка. Таким образом величина смещения должна равняться высоте увеличенного рисунка, т. е. $S \cdot V_y$.

```
float Ty = S * Vy; // смещение в положительную сторону по оси Oy после смены знака
```

Теперь, все наши выдвинутые условия выполняются: рисунок увеличивается и уменьшается вместе с увеличением или уменьшением окна, оставаясь полностью видимым, а его левый верхний угол привязан к левому верхнему углу окна.



2.3 Изменение соотношения сторон рисунка

Наряду с тем режимом вывода рисунка, что мы уже реализовали, добавим возможность выводить рисунок с изменением его соотношения сторон до соотношения сторон окна. Будем менять режим вывода рисунка по нажатию клавиши **М**. Для этого заведем в классе формы приватный параметр `keepAspectRatio` с логическим значением. Когда его значение будет равно `true` будем выводить рисунок в уже реализованном режиме, а в противном случае — с измененным соотношением сторон.

В файле `MyForm.h` найдите строку

```
#pragma endregion
```

После этой строки вставим описание параметра `keepAspectRatio`.

```
private: bool keepAspectRatio; // значение - сохранять ли соотношение сторон рисунка?
```

После этого в конструкторе формы в панели свойств окна создайте процедуру, обработчик события `Load` (Загрузка). Эта процедура выполняется автоматически, при запуске приложения, перед началом всех отрисовок.

Добавим в эту процедуру инициализацию параметра `keepAspectRatio`.

```
keepAspectRatio = true; // начальное значение: сохранять соотношение сторон рисунка
```

Теперь вернемся в конструктор формы и добавим процедуру обработчик события `KeyDown` (Нажата клавиша). В созданной процедуре присутствует параметр `e` типа `System::Windows::Forms::KeyEventArgs^`, из которого, с помощью выражения `e->KeyCode`, можно извлечь информацию о том, какая клавиша была нажата.

Добавим оператор `switch` в котором действия будут зависеть от значения этого выражения:

```
switch(e->KeyCode) {
default:
    break;
}
```

В теле оператора добавим действия на нажатие клавиши **М**: изменить значение параметра `keepAspectRatio` на противоположное. Если нажата клавиша **М**, то значение `e->KeyCode` будет `System::Windows::Forms::Keys::M` (так как пространство имен `System::Windows::Forms::Keys::M` подключено, достаточно писать `Keys::M`).

```
switch (e->KeyCode) {
case Keys::M :
    keepAspectRatio = !keepAspectRatio;
    break;
default:
    break;
}
```

Не забываем, что после обработки нажатия клавиши нужно обновить изображение в окне. Таким образом, после оператора `switch` следует поставить вызов

```
Refresh();
```

Обработчик события *KeyDown* примет вид:

```
1 private: System::Void MyForm_KeyDown(System::Object^ sender, System::Windows::Forms::KeyEventArgs^ e) {
2     switch (e->KeyCode) {
3     case Keys::M :
4         keepAspectRatio = !keepAspectRatio;
5         break;
6     default:
7         break;
8     }
9     Refresh();
10 }
```

Вернемся к разработке обработчика события *Paint*.

Чтобы соотношение сторон рисунка совпадало с соотношением сторон окна нужно, чтобы коэффициенты увеличения по осям *Ox* и *Oy* были различными. Поэтому вместо коэффициента *S* создадим по одному коэффициенту для каждой из осей, которые обозначим *Sx* и *Sy*. Чтобы уже реализованная версия работала без изменений, присвоим этим коэффициентам одно и то же значение. Сохраним этот код, для случая истинного значения `keepAspectRatio`. В случае, когда `keepAspectRatio` равно `false` вычислим эти коэффициенты по отдельности. Не забываем, что в остальном коде нужно заменить *S* на *Sx* и *Sy* при вычислении соответствующих координат.

```
float Sx, Sy;
if (keepAspectRatio) {
    // коэффициенты увеличения при сохранении исходного соотношения сторон
    Sx = Sy = aspectFig < aspectForm ? Wy / Vy : Wx / Vx;
}
else {
    Sx = Wx / Vx; // коэффициент увеличения по оси Ox
    Sy = Wy / Vy; // коэффициент увеличения по оси Oy
}
float Ty = Sy * Vy; // смещение в положительную сторону по оси Oy после смены знака
for (int i = 0; i < arrayLength; i += 4) {
    g->DrawLine(blackPen, Sx * lines[i], Ty - Sy * lines[i + 1],
                Sx * lines[i + 2], Ty - Sy * lines[i + 3]);
}
```

После всех изменений получим следующий блок глобальных переменных.

```
1 float lines[] = {
2     // голова
3     0.5f, 3.f, 1.f, 4.5f, // от левой щеки вверх до уха
4     1.f, 4.5f, 0.5f, 6.f, // левое ухо слева снизу вверх
5     0.5f, 6.f, 0.5f, 7.5f, // левое ухо слева
6     0.5f, 7.5f, 1.f, 8.f, // левое ухо вверх слева
7     1.f, 8.f, 1.5f, 8.f, // левое ухо вверх середина
8     1.5f, 8.f, 2.f, 7.5f, // левое ухо вверх справа
```

```

9  2.f,7.5f,1.5f, 6.f, // левое ухо справа сверту вниз
10 1.5f, 6.f,1.5f,4.5f, // левое ухо справа до макушки
11 1.5f,4.5f,3.f,4.5f, // макушка
12 3.f,4.5f,3.f,6.f, // правое ухо слева снизу вверх
13 3.f,6.f,2.5f,7.5f, // правое ухо слева
14 2.5f,7.5f,3.f,8.f, // правое ухо вверх слева
15 3.f,8.f,3.5f,8.f, // правое ухо вверх середина
16 3.5f,8.f,4.f,7.5f, // правое ухо вверх справа
17 4.f,7.5f,4.f,6.f, // правое ухо сверту вниз
18 4.f,6.f,3.5f,4.5f, // правое ухо справа
19 3.5f,4.5f,4.f,3.f, // от правого уха вниз до щеки
20 4.f,3.f,3.5f,1.5f, // правая скула
21 3.5f,1.5f,2.5f,1.f, // подбородок справа
22 2.5f,1.f,2.f,1.f, // подбородок снизу
23 2.f,1.f,1.f,1.5f, // подбородок слева
24 1.f,1.5f,0.5f,3.f, // левая скула
25 // туловище
26 4.f,3.f,5.5f,3.5f, // спина от головы вправо
27 5.5f,3.5f,7.f,3.5f, // спина вверх
28 7.f,3.5f,7.5f,2.5f, // спина сверту до хвоста
29 7.5f,2.5f,8.f,2.5f, // хвост сверту
30 8.f,2.5f,8.f,2.f, // хвост справа
31 8.f,2.f,7.5f,2.f, // хвост низ справа налево
32 7.5f,2.f,7.5f,0.5f, // задняя нога справа сверту вниз
33 7.5f,0.5f,6.5f,0.5f, // задняя нога низ
34 6.5f,0.5f,6.5f,1.f, // задняя нога слева
35 6.5f,1.f,6.f,1.f, // между задних ног
36 6.f,1.f,6.f,0.5f, // левая задняя нога справа
37 6.f,0.5f,5.f,0.5f, // левая задняя нога низ
38 5.f,0.5f,5.f,1.f, // левая задняя нога слева
39 5.f,1.f,4.f,1.f, // между задними и передними ногами
40 4.f,1.f,4.f,0.5f, // правая передняя нога справа
41 4.f,0.5f,3.f,0.5f, // правая передняя нога низ
42 3.f,0.5f,3.f,1.f, // правая передняя нога слева
43 3.f,1.f,2.5f,1.f, // между передних ног
44 2.5f,1.f,2.5f,0.5f, // передняя нога справа
45 2.5f,0.5f,1.5f,0.5f, // передняя нога низ
46 1.5f,0.5f,1.5f,1.25f, // передняя нога слева
47 // левый глаз
48 1.5f,3.5f,1.5f,3.f, // левый глаз слева сверту вниз
49 1.5f,3.f,2.f,3.f, // левый глаз низ
50 2.f, 3.f,2.f,3.5f, // левый глаз справа
51 2.f,3.5f,1.5f,3.5f, // левый глаз вверх
52 // правый глаз
53 2.5f,3.5f,2.5f,3.f, // правый глаз слева
54 2.5f,3.f,3.f, 3.f, // правый глаз снизу
55 3.f,3.f,3.f,3.5f, // правый глаз справа
56 3.f,3.5f,2.5f,3.5f, // правый глаз сверту
57 // ушные раковины
58 1.f,5.5f,1.f,7.f, // левая ушная раковина
59 3.5f,5.5f,3.5f,7.f, // правая ушная раковина
60 // нос
61 2.f,2.5f,2.5f,2.5f, // нос сверту
62 2.5f,2.5f,2.25f,2.f, // нос справа
63 2.25f,2.f,2.f,2.5f // нос слева
64 };
65 unsigned int arrayLength = sizeof(lines) / sizeof(float); // длина массива
66 float Vx = 8.5f; // размер рисунка по горизонтали
67 float Vy = 8.5f; // размер рисунка по вертикали
68 float aspectFig = Vx / Vy; // соотношение сторон рисунка

```

Обработчик события *Paint* примет следующий вид.

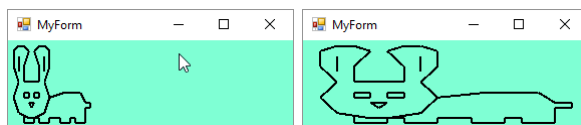
```

1 private: System::Void MyForm_Paint(System::Object^ sender, System::Windows::Forms::PaintEventArgs^ e) {
2     Graphics^ g = e->Graphics;
3     g->Clear(Color::Aquamarine);
4
5     Pen^ blackPen = gcnew Pen(Color::Black, 2);
6
7     float Wx = ClientRectangle.Width; // размер окна по горизонтали
8     float Wy = ClientRectangle.Height; // размер окна по вертикали
9
10    float aspectForm = Wx / Wy; // соотношение сторон окна рисования
11    float Sx, Sy;
12    if (keepAspectRatio) {
13        // коэффициенты увеличения при сохранении исходного соотношения сторон
14        Sx = Sy = aspectFig < aspectForm ? Wy / Vy : Wx / Vx;
15    }
16    else {
17        Sx = Wx / Vx; // коэффициент увеличения по оси Ox
18        Sy = Wy / Vy; // коэффициент увеличения по оси Oy
19    }
20    float Ty = Sy * Vy; // смещение в положительную сторону по оси Oy после смены знака
21    for (int i = 0; i < arrayLength; i += 4) {
22        g->DrawLine(blackPen, Sx * lines[i], Ty - Sy * lines[i + 1], Sx * lines[i + 2], Ty - Sy * lines[i + 3]);
23    }
24 }

```

Сразу после запуска приложения изменения не будут заметны. Но нажатие клавиши **M**

приведет к изменению соотношения сторон выведенного рисунка.



2.4 Задание для самостоятельной работы

Задание 2

1. Создайте приложение, включающее в себя все, что было описано выше в качестве примера. Проект должен называться Вашей фамилией, записанной латинскими буквами.
2. Выберите для себя свободный вариант задания 2 на портале course.sgu.ru (Ссылка «Выбор варианта для задания 2»).
3. Добавьте в получившийся проект следующее дополнение. Назначьте реакцию на нажатие клавиши **N**, приводящую к смене изображения построенного примера на рисунок, соответствующий Вашему варианту, и наоборот.



Элементы изображения, соответствующего Вашему варианту, Вам понадобится в последующих заданиях. Поэтому настоятельно Вам рекомендуем делать комментарии к координатам подобно комментариям в примере.



Для выполнения задания Вам нужно определить дополнительный набор глобальных переменных, по аналогии с приведенным примером. После этого, в обработчике события *Paint* стоит описать подобные переменные, которым следует присваивать те или иные глобальные значения, в зависимости от того, какое изображение следует выводить.



В некоторых изображениях присутствуют закрашенные многоугольники. В этом задании перед вами не стоит задача закраски изображения. Вашей задачей является только отрисовка всех ребер изображения, включая контуры закрашенных многоугольников.

4. Архив получившегося проекта загрузите на портал как ответ на задание 2 (см. раздел 1.7 о составе архива).



2.5 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Как задана система координат в окне? Где находится начало координат? Куда направлены координатные оси?
2. Система координат в окне правая или левая?
3. Система координат, связанную с рисунком, которую мы ввели и используем в разделе 2.2, правая или левая?
4. Что изменилось бы, если бы мы массив `lines` объявили локальным для обработчика событий *Paint*?
5. Что обозначают глобальные переменные `Vx` и `Vy` ? Почему эти переменные мы объявили глобальными?
6. Объясните смысл значений `Sx` , `Sy` и `Ty` .
7. Почему при отрисовке отрезка в обработчике события *Paint* координаты `y`, извлекаемые из массива `lines` , домножаются на отрицательные числа?
8. В чем смысл сравнения значений переменных `aspectForm` и `aspectFig` ?
9. Что обозначает переменная `keepAspectRatio` ?
10. Как назначить реакцию на нажатие клавиши для приложения `Windows Forms`?



3. Преобразования изображений

3.1 Предварительная подготовка

За основу проекта для третьего задания возьмем уже готовый проект, получившийся в результате выполнения второго задания.

Сделайте копию папки с проектом второго задания (чтобы не повторять предварительные установки). Имя проекта должно остаться без изменений.

В рамках этого задания Вы продолжите работать с рисунком, соответствующим Вашему варианту в задании 2. В ходе построения решения мы будем изменять те или иные готовые фрагменты, поэтому предварительно производить «очистку» проекта не следует.

3.2 Предмет реализации

Мы перестроим проект следующим образом.

1. Определим реакции на нажатия клавиш, которые приводят к изменению вывода изображения в окне: реализуем основные преобразования изображения, такие как поворот, масштабирование, сдвиг и др.
2. Изменим формат представления в программе объектов для изображения.
3. Реализуем чтение исходных данных изображения из файла.

3.3 Геометрические преобразования

Реализуем элементарные геометрические преобразования нашего изображения. Назначим реакции на нажатия клавиш, приводящие к тем или иным преобразованиям. Каждое преобразование будет применяться ко всем точкам изображения (т. е. ко всему изображению сразу).

Преобразования будем представлять в матричной форме. Для этого, каждую точку с координатами (x, y) переведем в однородные координаты: $(x, y, 1.0)$. Определим матрицу начального преобразования `initT`, включающую в себя масштабирование и сдвиг, которые мы определили в решении предыдущего задания. Тогда, если нужно изобразить отрезок

от точки A до точки B (где каждая точка представлена вектором-столбцом однородных координат), будем изображать отрезок от $\text{initT} \cdot A$ до $\text{initT} \cdot B$.

Каждое последующее преобразование (на нажатия клавиш) будем представлять своей матрицей T_i . Получится, что первоначальное изображение после нажатия клавиш будет состоять из отрезков от $T_k \cdot \dots \cdot T_2 \cdot T_1 \cdot \text{initT} \cdot A$ до $T_k \cdot \dots \cdot T_2 \cdot T_1 \cdot \text{initT} \cdot B$.

Если обозначим матрицу $T_k \cdot \dots \cdot T_2 \cdot T_1 \cdot \text{initT}$ через T , то в этих обозначений наши отрезки должны быть от $T \cdot A$ до $T \cdot B$.

Тогда получаем следующий алгоритм:

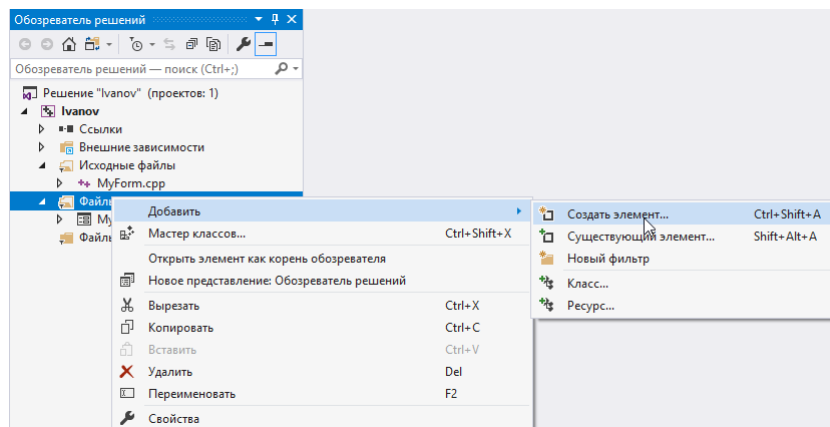
- для получения начального изображения присвоим $T = \text{initT}$ и отрисуем все отрезки от $T \cdot A$ до $T \cdot B$.
- после каждого нажатия клавиши, которой соответствует преобразование, выраженное матрицей T_i , присвоим $T = T_i \cdot T$ и отрисуем все отрезки от $T \cdot A$ до $T \cdot B$.

Таким образом, в процедуре отрисовки мы всегда будем изображать отрезки от $T \cdot A$ до $T \cdot B$. Начальное значение матрицы T мы присвоим при инициализации изображения, а в обработчике события *KeyDown* мы будем изменять матрицу T . Но, прежде чем перейти к реализации этого, нам понадобится реализовать матричные операции.

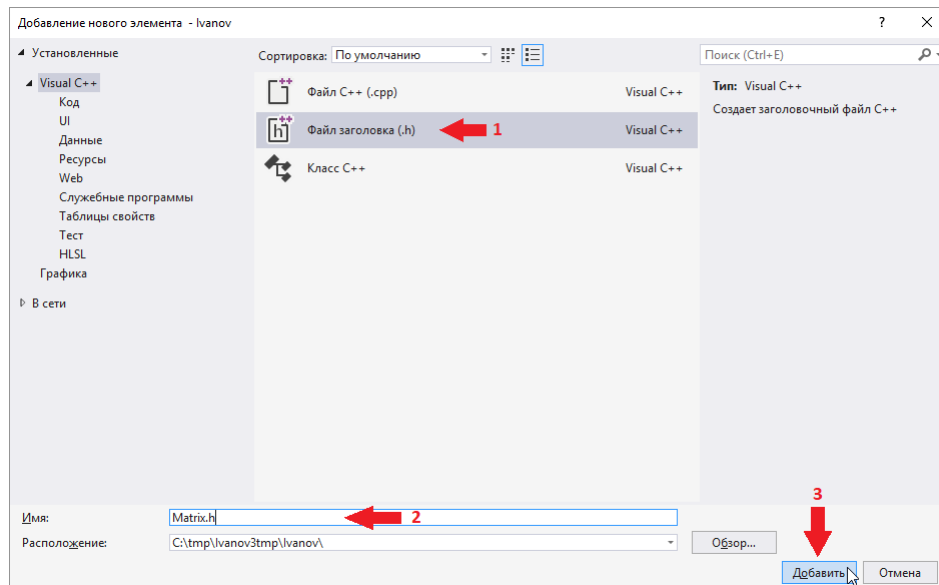
3.3.1 Реализация матричных операций

Для представления двумерных точек нам понадобятся векторы размерности 2, для двумерных точек в однородных координатах понадобятся векторы размерности 3. Кроме того, нам нужно каким то образом представить матрицы и операции умножения матрицы на матрицу и на вектор (в последующих заданиях нам могут понадобиться другие операции с матрицами и векторами, которые мы реализуем по мере необходимости).

В обозревателе решений нажмите правую кнопку мыши на папке *Файлы заголовков*. В появившемся меню выберите *Добавить* → *Создать элемент*.



В окне добавления нового элемента остается выбрать пункт *Файл заголовка (.h)* (пункт 1 на рисунке). В качестве имени файла укажите *Matrix.h* (пункт 2) и нажмите *Добавить* (пункт 3).



Будет создан файл заголовка с единственной строкой `#pragma once`. Все изменения мы будем вносить после этой строки.

Сначала создадим класс `vec2` для векторов размерности 2. Составляющими экземпляра данного класса будут две вещественные координаты `x` и `y`.

```
class vec2 {
public:
    float x, y;
    vec2() {}
};
```

После стандартного конструктора без аргументов добавим еще один, с двумя вещественными параметрами.

```
vec2(float a, float b) : x(a), y(b) {}
```

Таким образом, класс `vec2` будет выглядеть следующим образом.

```
1 class vec2 {
2 public:
3     float x, y;
4     // конструкторы
5     vec2() {}
6     vec2(float a, float b) : x(a), y(b) {}
7 };
```

По аналогии определим класс `vec3` для векторов размерности 3. По сравнению с предыдущим кодом, в коде для `vec3` в третьей строке после `x` и `y` должна появиться координата `z`, и подобное изменение в строке 6.

Тип `vec3` — основной рабочий тип элементов (точек) в наших вычислениях. Поэтому в классе `vec3` определим еще несколько операций.

Во первых, для упрощения перехода от евклидовых координат к однородным, добавим еще один конструктор, в котором первый аргумент — двумерный вектор, а второй — дополнительная третья координата.

```
vec3(vec2 v, float c) : vec3(v.x, v.y, c) {}
```

Определим операцию `*` для элементов `vec3` как результат покоординатного произведения исходных векторов

$$(x_1, y_1, z_1) * (x_2, y_2, z_2) = (x_1 \cdot x_2, y_1 \cdot y_2, z_1 \cdot z_2).$$

Такая операция не имеет ничего общего с векторным или скалярным произведением векторов, но она пригодится нам в дальнейшем.

Сначала в классе `vec3` перегрузим оператор `*=`, добавив после описания конструкторов следующий код

```
vec3& operator*=(const vec3& v) {
    x *= v.x;
    y *= v.y;
    z *= v.z;
    return *this;
}
```

Теперь, пользуясь оператором `*=`, перегрузим операцию `*`

```
const vec3 operator*(const vec3& v) {
    return vec3(*this) *= v; // делаем временную копию текущего объекта,
                             // которую домножаем на данный вектор,
                             // и возвращаем ее как результат
}
```

Для того, чтобы к элементам вектора можно было обратиться по индексу, переопределим оператор `[]`, добавив в определение класса следующий код:

```
float& operator[](int i) {
    return ((float*)this)[i]; // ссылку на текущий объект рассматриваем как ссылку
                              // на нулевой элемент массива значений типа float,
                              // после чего обращаемся к его i-му элементу
}
```

Общий вид описания класса `vec3` должен получиться следующий:

```
1 class vec3 {
2 public:
3     float x, y, z;
4     // конструкторы
5     vec3() {}
6     vec3(float a, float b, float c) : x(a), y(b), z(c) {}
7     vec3(vec2 v, float c) : vec3(v.x, v.y, c) {}
8     // умножение векторов *= и *
9     vec3& operator*=(const vec3& v) {
10         x *= v.x;
11         y *= v.y;
12         z *= v.z;
13         return *this;
14     }
15     const vec3 operator*(const vec3& v) {
16         return vec3(*this) *= v; // делаем временную копию текущего объекта,
17                                   // которую домножаем на данный вектор,
18                                   // и возвращаем ее как результат
19     }
```

```

20 // перегрузка []
21 float& operator[](int i) {
22     return ((float*)this)[i]; // ссылку на текущий объект рассматриваем как ссылку
23                               // на нулевой элемент массива значений типа float,
24                               // после чего обращаемся к его i-му элементу
25 }
26 };

```

После описания класса `vec3` добавим описание операции скалярного произведения трехмерных векторов `v1` и `v2`. Это можно реализовать как вычисление суммы координат результата операции `v1 * v2`.

```

float dot(vec3 v1, vec3 v2) {
    vec3 tmp = v1 * v2; // вычисляем произведения соответствующих координат
    return tmp.x + tmp.y + tmp.z; // и возвращаем их сумму
}

```

Теперь мы готовы описать класс `mat3` для представление матриц 3×3 . Представим матрицу как набор из трех строк — трех векторов размерности 3.

```

class mat3 {
public:
    vec3 row1, row2, row3;
    mat3() {}
};

```

Добавим несколько дополнительных конструкторов.

Пусть первый получает в качестве аргументов три вектора (по аналогии с конструкторами для векторов).

```

mat3(vec3 r1, vec3 r2, vec3 r3) : row1(r1), row2(r2), row3(r3) {}

```

Еще один — конструктор для создания диагональной матрицы с одинаковым числом на главной диагонали.

```

mat3(float a) {
    row1 = vec3(a, 0.f, 0.f);
    row2 = vec3(0.f, a, 0.f);
    row3 = vec3(0.f, 0.f, a);
}

```

Перегрузим оператор `[]`, опять же, по аналогии с перегрузкой для `vec3`.

```

vec3& operator[](int i) {
    return ((vec3*)this)[i]; // массив значений типа vec3
}

```

Когда мы перегрузили операторы `[]` для векторов и для матриц, можем с к элементам матрицы обращаться как к элементам двумерного массива. Пользуясь этим определим метод `transpose` для транспонирования матрицы (применение `A.transpose()` изменяет значение `A` на транспонированную матрицу, после чего выдает это значение).

```

mat3 transpose() {
    mat3 tmp(*this); // делаем временную копию матрицы
    for (int i = 0; i < 3; i++)

```

```

    for (int j = 0; j < 3; j++)
        (*this)[i][j] = tmp[j][i]; // заменяем элементы текущего объекта
                                   // из временной копии
    return *this;
}

```

Теперь реализуем операцию умножения матрицы на вектор. Для этого произведем перегрузку оператора `*`. Для определения операции вспомним, что результат произведения матрицы на вектор-столбец — это вектор, каждый элемент которого вычисляется как скалярное произведение соответствующей строки матрицы с заданным вектором.

$$\begin{bmatrix} row_1 \\ row_2 \\ row_3 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} row_1 \cdot v_1 \\ row_2 \cdot v_2 \\ row_3 \cdot v_3 \end{bmatrix}$$

Это можно записать следующим образом.

```

const vec3 operator* (const vec3 &v) {
    vec3 res = new(vec3); // создаем новый вектор (для результата)
    for (int i = 0; i < 3; i++) {
        (*res)[i] = dot((*this)[i], v); // i-й элемент вектора - скалярное произведение
    }
    return *res;
}

```

Умножение матриц можно представить следующим образом: каждый столбец матрицы-результата получается в результате умножения первой матрицы на соответствующий столбец второй.

$$M_1 \cdot \begin{bmatrix} col_1 & col_2 & col_3 \end{bmatrix} = \begin{bmatrix} M_1 \cdot col_1 & M_1 \cdot col_2 & M_1 \cdot col_3 \end{bmatrix}$$

Таким образом, нам нужно перебрать столбцы второй матрицы. Но так как наши матрицы хранятся по строкам, будем работать с транспонированной матрицей: каждая строка транспонированной матрицы есть соответствующий столбец исходной. Поэтому наш алгоритм для произведения матриц A и B будет следующим:

- транспонируем матрицу B , получим B^T ;
- перемножим матрицу A с каждой строкой B^T , результаты объединим в матрицу C ;
- результат — матрица C^T , полученная транспонированием матрицы C .

Перегрузим оператор `*=`, реализовав этот алгоритм

```

mat3& operator*= (const mat3 &m) {
    mat3 A(*this), B(m); // создаем копии исходных матриц
    B.transpose();        // транспонируем вторую матрицу
    for (int i = 0; i < 3; i++)
        (*this)[i] = A * B[i]; // в i-ю строку текущего объекта записываем
                                // результат перемножения первой матрицы с i-й строкой
                                // транспонированной матрицы,
    return (*this).transpose(); // транспонируем текущий объект, получаем результат
}

```

Теперь, используя последнюю процедуру, можем перегрузить оператор `*` (по примеру аналогичных перегрузок для векторов).

```

const mat3 operator* (const mat3 &m) {
    return mat3(*this) *= m;
}

```

В результате получим следующий состав класса `mat3`

```

1 class mat3 {
2 public:
3     vec3 row1, row2, row3;
4     // конструкторы
5     mat3() {}
6     mat3(vec3 r1, vec3 r2, vec3 r3) : row1(r1), row2(r2), row3(r3) {}
7     mat3(float a) {
8         row1 = vec3(a, 0.f, 0.f);
9         row2 = vec3(0.f, a, 0.f);
10        row3 = vec3(0.f, 0.f, a);
11    }
12    // перегрузка []
13    vec3& operator[](int i) {
14        return ((vec3*)this)[i]; // массив значений типа vec3
15    }
16    // транспонирование
17    mat3 transpose() {
18        mat3 tmp(*this); // делаем временную копию матрицы
19        for (int i = 0; i < 3; i++)
20            for (int j = 0; j < 3; j++)
21                (*this)[i][j] = tmp[j][i]; // заменяем элементы текущего объекта
22                // из временной копии
23        return *this;
24    }
25    // умножение матрицы на вектор
26    const vec3 operator* (const vec3 &v) {
27        vec3* res = new(vec3); // создаем новый вектор (для результата)
28        for (int i = 0; i < 3; i++) {
29            (*res)[i] = dot((*this)[i], v); // i-й элемент вектора - скалярное произведение
30        }
31        return *res;
32    }
33    // произведение матриц *= u =
34    mat3& operator*= (const mat3 &m) {
35        mat3 A(*this), B(m); // создаем копии исходных матриц
36        B.transpose(); // транспонируем вторую матрицу
37        for (int i = 0; i < 3; i++)
38            (*this)[i] = A * B[i]; // в i-ю строку текущего объекта записываем
39            // результат перемножения первой матрицы с i-й строкой
40            // транспонированной матрицы,
41        return (*this).transpose(); // транспонируем текущий объект, получаем результат
42    }
43    const mat3 operator* (const mat3 &m) {
44        return mat3(*this) * m;
45    }
46 };

```

И в завершение, опишем функцию `normalize`, получающую для вектора типа `vec3` вектор `vec2`, организовав переход из однородных координат в евклидовы (разделив первые две координаты на третью).

```

vec2 normalize(vec3 v) {
    return vec2(v.x / v.z, v.y / v.z);
}

```

На текущем этапе разработка `Matrix.h` завершена.

Подключим наш файл к проекту. Из обозревателя решений откройте файл `MyForm.cpp`.

Перед строкой

```
#include "MyForm.h"
```

вставьте строку

```
#include "Matrix.h"
```

Можно запустить проект на компиляцию и выполнение. Если в файле `Matrix.h` ошибок не допущено, то проект должен работать так же, как и раньше.

3.3.2 Элементарные преобразования

По примеру, описанному в предыдущем разделе, добавим в проект еще один файл заголовка, который назовем `Transform.h`. В этом файле определим процедуры для элементарных преобразований, из которых будем строить все наши преобразования в проекте.

Так как мы будем использовать операции, реализованные в `Matrix.h`, подключим этот файл.

```
#include "Matrix.h"
```

Нам понадобятся функции вычисления синуса и косинуса. Поэтому следует подключить еще и `math.h`.

```
#include <math.h>
```

Для каждого преобразования напомним функцию, возвращающую матрицу этого преобразования.

Начнем с преобразования переноса. Как Вы должны знать, матрица переноса в двумерной графике определяется двумя параметрами T_x и T_y . Параметры являются величинами сдвига вдоль соответствующих осей координат, по отношению к исходному изображению. Матрицу переноса можно получить из единичной матрицы, заполнив её последний столбец этими параметрами.

Это можно реализовать следующей процедурой.

```
mat3 translate(float Tx, float Ty) {  
    mat3 *res = new mat3(1.f); // создали единичную матрицу  
    (*res)[0][2] = Tx; // поменяли  
    (*res)[1][2] = Ty; // значения в последнем столбце  
    return *res;  
}
```

Преобразование масштабирования также зависит от двух коэффициентов S_x и S_y . Аналогично, создадим матрицу преобразования масштабирования из единичной матрицы. Только в этом случае параметры масштабирования замещают собой элементы матрицы на главной диагонали.

```
mat3 scale(float Sx, float Sy) {  
    mat3 *res = new mat3(1.f); // создали единичную матрицу  
    (*res)[0][0] = Sx; // поменяли  
    (*res)[1][1] = Sy; // значения на главной диагонали  
    return *res;  
}
```

Реализуем еще одну функцию, на случай когда $S_x = S_y = S$. Воспользуемся для этого уже полученной функцией `scale`.

```
mat3 scale(float S) {
    return scale(S, S);
}
```

Наконец, разработаем функцию, выдающую матрицу поворота относительно начала координат против часовой стрелки на заданный угол. Матрица зависит от одного параметра — величины угла поворота `theta`. Первые два элемента на главной диагонали матрицы заполняются косинусами заданного угла, а соседние с ними элементы — синусами. Так как в окне формы система координат левая, то во второй строке матрицы знак при синусе — минус, а в первой — плюс. Получим следующую процедуру.

```
mat3 rotate(float theta) {
    mat3 *res = new mat3(1.f); // создали единичную матрицу
    (*res)[0][0] = (*res)[1][1] = (float)cos(theta); // заполнили главную диагональ
    (*res)[0][1] = (float)sin(theta); // синус в первой строке (с плюсом)
    (*res)[1][0] = -(*res)[0][1]; // синус во второй строке (с минусом)
    return *res;
}
```

Пока, в разработке файла `Transform.h` остановимся на достигнутом.

Подключим наш файл к проекту. В файле `MyForm.cpp`. Перед строкой

```
#include "MyForm.h"
```

вставьте строку

```
#include "Transform.h"
```

Можно запустить проект на компиляцию и выполнение. Если в файле `Transform.h` ошибок не допущено, то проект должен работать так же, как и раньше.

3.3.3 Начальная отрисовка в матричной форме

Как мы уже говорили в начале раздела 3.3, нам понадобятся две вспомогательные матрицы `T` и `initT`, с помощью которых мы будем преобразовывать изображение на форме. Опишем их в блоке описания глобальных переменных.

```
mat3 T = mat3(1.f); // матрица, в которой накапливаются все преобразования
                      // первоначально - единичная матрица
mat3 initT; // матрица начального преобразования
```

Будем здесь предполагать, что в результате второго задания в процедуре `MyForm_Paint` присутствует фрагмент, в котором производится отрисовка всех имеющихся отрезков¹:

```
for (int i = 0; i < arrayLength; i += 4) {
    g->DrawLine(blackPen, Sx * lines[i], Ty - Sy * lines[i + 1]
                , Sx * lines[i + 2], Ty - Sy * lines[i + 3]);
}
```

¹Если Ваша версия обработчика события `Paint` отличается значительно, Вы можете вернуться к версии этой процедуры (и блока глобальных переменных, отвечающих за набор отрезков), полученной в конце главы 2

Если внимательно посмотреть на вызов `DrawLine`, то можно сделать представление о том преобразовании, что применяется к каждой точке. Это преобразование и запишем в матрицу `initT`. Рассмотрим этот момент поподробней:

- каждая координата x домножается на Sx , а каждая координата y — на $-Sy$, т.е. происходит масштабирование с этими параметрами;
- после масштабирования происходит перенос по оси Oy на Ty , а по оси Ox сдвиг отсутствует или, по другому, по оси Ox происходит сдвиг на $0.f$.

Перепишем этот фрагмент несколько иначе, подключив матричную форму.

```
initT = translate(0.f, Ty) * scale(Sx, -Sy); // преобразования применяются справа налево
// сначала масштабирование, а потом перенос
// в initT совмещаем эти два преобразования
mat3 M = T * initT; // совмещение начального преобразования и
// накопленных преобразований
for (int i = 0; i < arrayLength; i += 4) {
    vec3 A = vec3(lines[i], lines[i + 1], 1.f); // начало отрезка в однородных координатах
    vec3 B = vec3(lines[i + 2], lines[i + 3], 1.f); // конец отрезка в однородных координатах
    vec2 a = normalize(M * A); // начало отрезка после преобразования
    vec2 b = normalize(M * B); // конец отрезка после преобразования
    g->DrawLine(blackPen, a.x, a.y, b.x, b.y); // отрисовка отрезка
}
```

Проведенные исправления не должны привести к каким-то изменениям в функционировании программы.

3.3.4 Назначение горячих клавиш

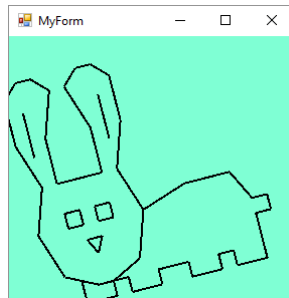
В обработчике события `KeyDown` добавим несколько дополнительных реакций на нажатия клавиш. Но прежде, перед оператором `switch` добавим вычисление координат центра текущего окна.

```
float Wcx = ClientRectangle.Width / 2.f; // координаты центра
float Wcy = ClientRectangle.Height / 2.f; // текущего окна
```

В операторе `switch` добавим реакцию на клавишу **Q** — поворот изображения вокруг центра окна на 0.01 радиана. Зная значения Wx и Wy можем легко найти координаты середины окна, а само преобразование будет заключаться в переносе начала координат в середину окна, повороте и обратном переносе. Это преобразование совместим с тем, что уже накопилось в `T`, т.е. домножим матрицу `T` слева последовательно на матрицы этих преобразований.

```
case Keys::Q:
    T = translate(-Wcx, -Wcy) * T; // перенос начала координат в (Wcx, Wcy)
    T = rotate(0.01f) * T; // поворот на 0.01 радиан относительно
    // нового центра
    T = translate(Wcx, Wcy) * T; // перенос начала координат обратно
    break;
```

Если все проведено корректно, то после запуска проекта нажатие клавиши **Q** будет приводить к повороту рисунка против часовой стрелки.



Добавим ещё одно преобразование на нажатие клавиши **W** — сдвиг изображения вверх на 1 пиксел

```
case Keys::W:
    T = translate(0.f, -1.f) * T; // сдвиг вверх на один пиксел
    break;
```

и на клавишу **Escape** — сброс всех сделанных преобразований.

```
case Keys::Escape:
    T = mat3(1.f); // присвоили T единичную матрицу
    break;
```

Запустите проект и опробуйте новые назначенные клавиши.

3.4 Внутренний формат представления объектов

Объект, который мы рисуем, состоит из набора отрезков. При создании этого набора нам пришлось столкнуться с тем, что большинство точек, упоминаемых в нем, повторяются дважды (а возможно и большее количество раз). Например, как конец одного отрезка и начало другого, инцидентного с ним, отрезка. Удобнее было бы представить изображение как набор ломаных линий, где каждая линия задана последовательностью точек. Более того, хотелось бы задать характеристики каждой такой ломаной, такие как цвет и толщина линии.

Опишем в отдельном заголовочном файле структуру данных для такого объекта. Создадим в проекте еще один заголовочный файл (в начале раздела 3.3.1 оговаривалось, как это сделать), который назовите `Figure.h`.

Координаты каждой точки ломаной будем представлять в виде двумерного вектора. Поэтому понадобится подключение файла заголовка с описанием типа `vec2`. Кроме того, саму ломаную будем представлять в виде набора таких точек объединенных в контейнер `vector` библиотеки STL. Для подключения добавим следующий код.

```
#include "Matrix.h"
#include <vector>
```

Далее, опишем класс `path`, в который включим то, о чем мы говорили.

```
class path {
public:
    std::vector<vec2> vertices; // последовательность точек
    vec3 color; // цвет, разбитый на составляющие RGB
    float thickness; // толщина линии
};
```

Здесь цвет линии представлен трехмерным вектором. Будем задавать цвет в формате RGB (в порядке следования в векторе), где каждая составляющая представлена числом от 0 до 255.

Добавим простой конструктор от трех аргументов, который будет объединять все заданные аргументы в объект.

```
path(std::vector<vec2> verts, vec3 col, float thickn) {
    vertices = verts;
    color = col;
    thickness = thickn;
}
```

Получим следующий вид файла Figure.h

```
1 #pragma once
2 #include "Matrix.h"
3 #include <vector>
4
5 class path {
6 public:
7     std::vector<vec2> vertices; // последовательность точек
8     vec3 color; // цвет, разбитый на составляющие RGB
9     float thickness; // толщина линии
10    path(std::vector<vec2> verts, vec3 col, float thickn) {
11        vertices = verts;
12        color = col;
13        thickness = thickn;
14    }
15 };
```

Подключим созданный файл заголовка в проект, добавив в файл MyForm.cpp перед строкой с `#include "MyForm.h"` строку

```
#include "Figure.h"
```

Проект должен запускаться без изменений.

Дальнейшие изменения коснутся содержимого файла MyForm.h, но прежде чем перейти к его редактированию, добавим в MyForm.cpp перед строкой с `#include "MyForm.h"`

```
#include <vector>
```

Перейдем к коду MyForm.h.

Чтобы каждый раз не писать префикс `std::` подключаем пространство имен `std`, добавив строку

```
using namespace std;
```

в блок аналогичных строк перед блоком описания глобальных переменных.

Будем предполагать, что мы работаем только с одним изображением. Изображение состоит из ломаных линий. Будем хранить эти линии в глобальной переменной — контейнере типа `vector`. Добавим её описание в блоке описания глобальных переменных

```
vector<path> figure;
```

Для примера, в обработчике события *Load* заполним список *figure* некоторыми значениями. Скопируйте отсюда следующий код.

```
float thickness;
vec3 color;
vector<vec2> vertices;
// голова
thickness = 2; // толщина линии 2
color = vec3(255,0,0); // цвет красный
// точки
vertices.push_back(vec2(0.5f, 3.f));
vertices.push_back(vec2(1.f, 4.5f));
vertices.push_back(vec2(0.5f, 6.f));
vertices.push_back(vec2(0.5f, 7.5f));
vertices.push_back(vec2(1.f, 8.f));
vertices.push_back(vec2(1.5f, 8.f));
vertices.push_back(vec2(2.f, 7.5f));
vertices.push_back(vec2(1.5f, 6.f));
vertices.push_back(vec2(1.5f, 4.5f));
vertices.push_back(vec2(3.f, 4.5f));
vertices.push_back(vec2(3.f, 6.f));
vertices.push_back(vec2(2.5f, 7.5f));
vertices.push_back(vec2(3.f, 8.f));
vertices.push_back(vec2(3.5f, 8.f));
vertices.push_back(vec2(4.f, 7.5f));
vertices.push_back(vec2(4.f, 6.f));
vertices.push_back(vec2(3.5f, 4.5f));
vertices.push_back(vec2(4.f, 3.f));
vertices.push_back(vec2(3.5f, 1.5f));
vertices.push_back(vec2(2.5f, 1.f));
vertices.push_back(vec2(2.f, 1.f));
vertices.push_back(vec2(1.f, 1.5f));
vertices.push_back(vec2(0.5f, 3.f));
figure.push_back(path(vertices, color, thickness));
// левый глаз
thickness = 4; // толщина линии 4
color = vec3(0, 255, 0); // цвет зеленый
// точки
vertices.clear();
vertices.push_back(vec2(1.5f, 3.5f));
vertices.push_back(vec2(1.5f, 3.f));
vertices.push_back(vec2(2.f, 3.f));
vertices.push_back(vec2(2.f, 3.5f));
vertices.push_back(vec2(1.5f, 3.5f));
figure.push_back(path(vertices, color, thickness));
// правый глаз
// цвет и толщина те же
vertices.clear();
vertices.push_back(vec2(2.5f, 3.5f));
vertices.push_back(vec2(2.5f, 3.f));
vertices.push_back(vec2(3.f, 3.f));
vertices.push_back(vec2(3.f, 3.5f));
vertices.push_back(vec2(2.5f, 3.5f));
figure.push_back(path(vertices, color, thickness));
```

Теперь перейдем к изменению обработчика события *Paint*.

Опять обратимся к циклу, ответственному за вывод линий. Сейчас (после последних изменений) он выглядит так:

```

1 for (int i = 0; i < arrayLength; i += 4) {
2   vec3 A = vec3(vec2(lines[i], lines[i + 1]), 1.f); // начало отрезка в однородных координатах
3   vec3 B = vec3(lines[i + 2], lines[i + 3], 1.f); // конец отрезка в однородных координатах
4   vec2 a = normalize(M * A); // начало отрезка после преобразования
5   vec2 b = normalize(M * B); // конец отрезка после преобразования
6   g->DrawLine(blackPen, a.x, a.y, b.x, b.y); // отрисовка отрезка
7 }

```

Вместо этого цикла вставим отрисовку линий из списка `figure`. То есть, у нас должен быть цикл, пробегающий по всем ломаным, сохраненным в `figure`:

```

for (int i = 0; i < figure.size(); i++) {
  path lines = figure[i]; // lines - очередная ломаная линия
}

```

Внутри этого цикла для каждой ломаной мы должны определить перо, которым будем проводить отрисовку. Для получения цвета из трех составляющих используем функцию `System::Drawing::Color::FromArgb`.

```

Pen^ pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
pen->Width = lines.thickness;

```

Теперь нужно организовать цикл по всем отрезкам в `lines` или, скажем лучше, цикл пробегающий по всем точкам — концам отрезков. Конец первого отрезка — вторая точка в списке отрезков (точка, с индексом 1). У каждого отрезка, за исключением первого, начальная точка — это конечная точка предыдущего отрезка.

Перед отрисовкой каждую точку переводим в однородные координаты, домножаем на матрицу преобразований и возвращаем в обычные евклидовы координаты.

Таким образом, вырисовывается следующий цикл.

```

vec2 start = normalize(M * vec3(lines.vertices[0], 1.0)); // первая начальная точка
for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
  vec2 end = normalize(M * vec3(lines.vertices[j], 1.0)); // конечная точка
  g->DrawLine(pen, start.x, start.y, end.x, end.y); // отрисовка отрезка
  start = end; // конечная точка текущего отрезка становится начальной точкой следующего
}

```

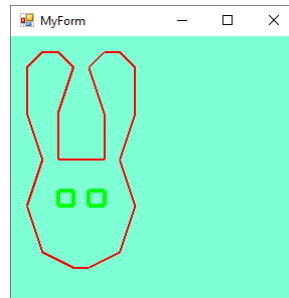
Весь цикл отрисовки должен выглядеть так:

```

1 for (int i = 0; i < figure.size(); i++) {
2   path lines = figure[i]; // lines - очередная ломаная линия
3   Pen^ pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
4   pen->Width = lines.thickness;
5   vec2 start = normalize(M * vec3(lines.vertices[0], 1.0)); // первая начальная точка
6   for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
7     vec2 end = normalize(M * vec3(lines.vertices[j], 1.0)); // конечная точка
8     g->DrawLine(pen, start.x, start.y, end.x, end.y); // отрисовка отрезка
9     start = end; // конечная точка текущего отрезка становится начальной точкой следующего
10  }
11 }

```

Если все введено без ошибок, то запуск приложения приведет к получению следующего изображения.



3.5 Файловый ввод данных

Безусловно, утомительно и некрасиво наполнять список `figure` набором команд подобным тому, что приведен в качестве примера в предыдущем разделе. Вместо этого организуем ввод исходных данных для изображения из текстового файла.

Сначала обговорим формат входного файла.

3.5.1 Формат входного файла

Для простоты исполнения, будем использовать следующий формат входного файла.

В файле могут встречаться пустые строки, а также строки комментариев. Строка комментариев должна начинаться с символа `#`.

Кроме пустых строк и строк с комментариями в файле могут встречаться строки команд следующего вида:

```
frame Vx Vy команда установки высоты и ширины изображения. Если команда встре-
чается в файле более одного раза, то каждый последующий экземпляр команды
отменяет действие предыдущей;
color R G B команда установки цвета для последующих объектов;
thickness p команда установки толщины линий последующих объектов;
path n задается ломаная линия состоящая из n точек. Команда объявляет, что в сле-
дующих строках (отличных от пустых и строк с комментариями) представлены
координаты n точек, по паре координат в отдельной строке.
```

Например, изображение, составленное нами в обработчике *Load*, можно задать файлом со следующим содержанием.

```
frame 8.5 8.5
# голова
color 255 0 0
thickness 2
path 23
# от левой щеки вверх
0.5 3.
1. 4.5
0.5 6.
# левое ухо вверх слева
0.5 7.5
1. 8.
1.5 8.
2. 7.5
1.5 6.
# макушка
1.5 4.5
```

```

3. 4.5
3. 6.
# правое ухо верт слева
2.5 7.5
3. 8.
3.5 8.
4. 7.5
4. 6.
3.5 4.5
# правая скула
4. 3.
3.5 1.5
2.5 1.
2. 1.
# левая скула
1. 1.5
0.5 3.

# глаза
color 0 255 0
thickness 4

# левый глаз
path 5
1.5 3.5
1.5 3.
2. 3.
2. 3.5
1.5 3.5

# правый глаз
path 5
2.5 3.5
2.5 3.
3. 3.
3. 3.5
2.5 3.5

```

Сохраните содержимое этого примера в текстовом файле с именем `Hare.txt`.



Обратите, пожалуйста, внимание, что сохранить текстовый файл желательно в кодировке `cp1251`. Можно сохранить файл и в другой кодировке, но тогда оставляйте первую строку файла пустой.

В дальнейшем, чтобы не предусматривать в коде дополнительный анализ, будем считать, что файл задан всегда корректно.

3.5.2 Чтение входного файла

Добавим в нашу форму загрузку данных из файла.

Сначала в файле `MyForm.cpp` перед строкой с `#include "MyForm.h"` добавим

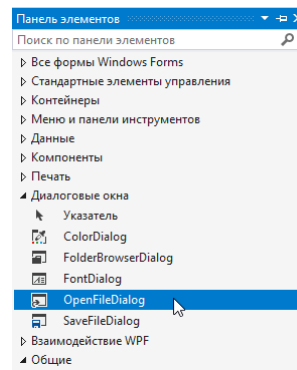
```

#include <fstream>
#include <sstream>

```

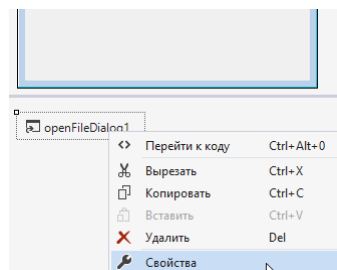
Будем определять имя файла для открытия с помощью диалогового окна открытия файлов. Для этого добавим на форму элемент `OpenFileDialog`. Перейдем в конструктор

формы, откроем панель элементов и в разделе *Диалоговые окна* схватите левой кнопкой мыши элемент *OpenFileDialog* и перетащите его на форму.



Он отобразится не на самой форме, а снизу от нее.

Кликните правой кнопкой мыши на этом элементе и откройте панель его свойств.



Поменяйте его атрибуты:

- *(Name)* = *openFileDialog*
- *(FileName)* Сделать поле пустым
- *Title* = *Открыть файл*
- *DefaultExt* = *txt*
- *Filter* = *Текстовые файлы (*.txt)|*.txt|Все файлы (*.*)|*.**

Кроме этого добавим на форму элемент *Button* (из раздела *Стандартные элементы управления*) и установим его атрибуты

- *(Name)* = *btnOpen*
- *Text* = *Открыть*

Так как мы добавили элемент на форму, то реакции на нажатия клавиш, которые мы определили, перестанут работать: мы добавляли эти реакции к окну формы (не к кнопке), а при наличии кнопки на форме фокус будет оставаться на кнопке. Чтобы это исправить, в конструкторе формы перейдите к свойствам формы и установите значение атрибута *KeyPreview* на *True*.

Для кнопки *btnOpen* добавим обработчик события *Click* (для этого достаточно дважды кликнуть по кнопке в конструкторе).

В коде обработчика сначала передаем управление диалоговому окну открытия файла `openFileDialog->ShowDialog()`. Этот вызов приведет к появлению окна для выбора файла пользователем. Если пользователь выберет имя файла и нажмет в этом окне кнопку

ОК, это будет означать, что нам нужно начать чтение из выбранного файла, а в противном случае — ничего не делать.

Если пользователь в диалоговом окне нажмет кнопку ОК, то вызов `openFileDialog->ShowDialog()` вернет значение

`System::Windows::Forms::DialogResult::OK`.

Поэтому проверку такого условия можно выразить условным оператором

```
if (openFileDialog->ShowDialog() == System::Windows::Forms::DialogResult::OK) {  
    // в файловом диалоге нажата кнопка ОК  
}
```

Если условие выполняется, то организуем чтение из выбранного файла. При этом будем использовать стандартный потоковый ввод (соответствующие библиотеки для этого мы уже подключили).

От файлового диалога нам понадобится только имя выбранного файла `openFileDialog->FileName`, но преобразованное в формат аргумента процедуры открытия файла. Для этого преобразования добавим в тело `if` следующий код.

```
// перезапись имени файла из openFileDialog->FileName в fileName  
wchar_t fileName[1024]; // переменная, в которой посимвольно сохраним имя файла  
for (int i = 0; i < openFileDialog->FileName->Length; i++)  
    fileName[i] = openFileDialog->FileName[i];  
fileName[openFileDialog->FileName->Length] = '\\0';
```

Теперь откроем файл и проверим его успешное открытие

```
// объявление и открытие файла  
ifstream in;  
in.open(fileName);  
if (in.is_open()) {  
    // файл успешно открыт  
}
```

Если файл открыт успешно, очищаем список `figure` (на случай, если изображение уже было загружено) и описываем временные переменные, в которые будем сохранять данные, считанные из файла. При этом удобно ввести значения по умолчанию для толщины линии и для составляющих цвета.

```
figure.clear(); // очищаем имеющийся список ломаных  
// временные переменные для чтения из файла  
float thickness = 2; // толщина со значением по умолчанию 2  
float r, g, b; // составляющие цвета  
r = g = b = 0; // значение составляющих цвета по умолчанию (черный)
```

После этого читаем и обрабатываем каждую строку файла. После того, как файл считан, нам нужно обновить изображение в соответствии с прочитанными данными.

```
// непосредственно работа с файлом  
string str; // строка, в которую считываем строки файла  
getline(in, str); // считываем из входного файла первую строку  
while (in) { // если очередная строка считана успешно  
    // обрабатываем строку
```



```
// считываем очередную строку
getline(in, str);
}
Refresh();
```

Для каждой прочитанной строки сначала проводим проверку — не является ли она пустой строкой или строкой с комментариями.

```
if ((str.find_first_not_of(" \t\r\n") != string::npos) && (str[0] != '#')) {
    // прочитанная строка не пуста и не комментарий
}
}
```

Если строка таковой не является, то из нее будем производить чтение информации. Для этого сначала представляем ее в виде стандартного строкового потока.

```
stringstream s(str); // строковый поток из строки str
```

Каждая значащая строка файла начинается с имени команды (за исключением строк, следующих за командой `path`). Поэтому, читаем из потока имя команды:

```
string cmd; // переменная для имени команды
s >> cmd; // считываем имя команды
```

Последующие действия должны зависеть от того, какая команда была прочитана:

```
if (cmd == "frame") { // размеры изображения
}
else if (cmd == "color") { // цвет линии
}
else if (cmd == "thickness") { // толщина линии
}
else if (cmd == "path") { // набор точек
}
}
```

В случае, если прочитана команда `frame`, необходимо прочитать из потока 2 числа, которые можно сразу сохранить в глобальных (уже описанных) переменных `Vx` и `Vy`. После этого обновим значение соотношения сторон

```
s >> Vx >> Vy; // считываем глобальные значения Vx и Vy
aspectFig = Vx / Vy; // обновление соотношения сторон
```

Если прочитана команда `color`, то считываем из потока три составляющие цвета в заготовленные для этого переменные.

```
s >> r >> g >> b; // считываем три составляющие цвета
```

Если прочитана команда `thickness`, то считываем значение толщины.

```
s >> thickness; // считываем значение толщины
```

Обработка команды `path` является более сложной. Сначала опишем временный список для прочитанных точек.

```
vector<vec2> vertices; // список точек ломаной
```

Опишем и прочитаем из строки количество точек.

```
int N; // количество точек
s >> N;
```

Чтобы прочитать нужное количество точек, заведем еще одну строковую переменную, и организуем цикл, который считывает построчно файл. Считаем, что после этого цикла все точки прочитаны и находятся в списке `vertices`. Поэтому после цикла из этого списка и значений переменных `r`, `g`, `b` и `thickness` сформируем объект `path` и запишем его в список `figure`.

```
string str1; // дополнительная строка для чтения из файла
while (N > 0) { // пока не все точки считали
    getline(in, str1); // считываем в str1 из входного файла очередную строку
    // так как файл корректный, то на конец файла проверять не нужно
}
// все точки считаны, генерируем ломаную (path) и кладем ее в список figure
figure.push_back(path(vertices, vec3(r, g, b), thickness));
```

Внутри цикла, так же, как и раньше, проверим, что прочитанная строка не пуста и не является комментарием.

```
if ((str1.find_first_not_of(" \t\r\n") != string::npos) && (str1[0] != '#')) {
    // прочитанная строка не пуста и не комментарий
    // значит в ней пара координат
}
```

Если в строке пара координат, то прочитаем её, сформируем объект `vec2` и добавим его в список `vertices` и скорректируем счетчик непрочитанных точек.

```
float x, y; // переменные для считывания
stringstream s1(str1); // еще один строковый поток из строки str1
s1 >> x >> y;
vertices.push_back(vec2(x, y)); // добавляем точку в список
N--; // уменьшаем счетчик после успешного считывания точки
```

В результате процедура `btnOpen_Click` должна принять следующий вид.

```
1 private: System::Void btnOpen_Click(System::Object^ sender, System::EventArgs^ e) {
2     if (openFileDialog->ShowDialog() == System::Windows::Forms::DialogResult::OK) {
3         // в файловом диалоге нажата кнопка OK
4         // перезапись имени файла из openFileDialog->FileName в fileName
5         wchar_t fileName[1024]; // переменная, в которой посимвольно сохраним имя файла
6         for (int i = 0; i < openFileDialog->FileName->Length; i++)
7             fileName[i] = openFileDialog->FileName[i];
8         fileName[openFileDialog->FileName->Length] = '\0';
9
10        // объявление и открытие файла
11        ifstream in;
12        in.open(fileName);
13        if (in.is_open()) {
```

```

14 // файл успешно открыт
15 figure.clear(); // очищаем имеющийся список ломаных
16 // временные переменные для чтения из файла
17 float thickness = 2; // толщина со значением по умолчанию 2
18 float r, g, b; // составляющие цвета
19 r = g = b = 0; // значение составляющих цвета по умолчанию (черный)
20 string cmd; // строка для считывания имени команды
21 // непосредственно работа с файлом
22 string str; // строка, в которую считываем строки файла
23 getline(in, str); // считываем из входного файла первую строку
24 while (in) { // если очередная строка считана успешно
25     // обрабатываем строку
26     if ((str.find_first_not_of(" \t\r\n") != string::npos) && (str[0] != '#')) {
27         // прочитанная строка не пуста и не комментарий
28         stringstream s(str); // строковый поток из строки str
29         s >> cmd;
30         if (cmd == "frame") { // размеры изображения
31             s >> Vx >> Vy; // считываем глобальные значения Vx и Vy
32         }
33         else if (cmd == "color") { // цвет линии
34             s >> r >> g >> b; // считываем три составляющие цвета
35         }
36         else if (cmd == "thickness") { // толщина линии
37             s >> thickness; // считываем значение толщины
38         }
39         else if (cmd == "path") { // набор точек
40             vector<vec2> vertices; // список точек ломаной
41             int N; // количество точек
42             s >> N;
43             string str1; // дополнительная строка для чтения из файла
44             while (N > 0) { // пока не все точки считали
45                 getline(in, str1); // считываем в str1 из входного файла очередную строку
46                 // так как файл корректный, то на конец файла проверять не нужно
47                 if ((str1.find_first_not_of(" \t\r\n") != string::npos) && (str1[0] != '#')) {
48                     // прочитанная строка не пуста и не комментарий
49                     // значит в ней пара координат
50                     float x, y; // переменные для считывания
51                     stringstream s1(str1); // еще один строковый поток из строки str1
52                     s1 >> x >> y;
53                     vertices.push_back(vec2(x, y)); // добавляем точку в список
54                     N--; // уменьшаем счетчик после успешного считывания точки
55                 }
56             }
57             // все точки считаны, генерируем ломаную (path) и кладем ее в список figure
58             figure.push_back(path(vertices, vec3(r, g, b), thickness));
59         }
60     }
61     // считываем очередную строку
62     getline(in, str);
63 }
64 Refresh();
65 }
66 }
67 }

```

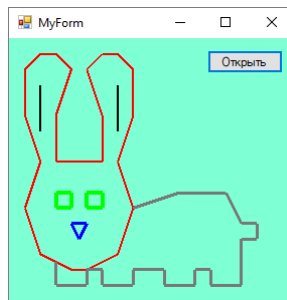
Если сейчас запустить проект, нажать на кнопку открытия файла и выбрать файл `Hare.txt`, то данные из файла будут считаны и изображение построено. Но сейчас изображение ничем не отличается от того, что было построено без загрузки файла.



Стоит обратить внимание на то, что принцип чтения из файла позволяет добавлять в наш файл незначущую информацию после параметров команд или координат точек, что может послужить возможностью добавлять комментарии к каждой строке входного файла.

На портале course.sgu.ru к заданию 3 приложен файл `Hare_full.txt`. Если этот файл

открыть в приложении, должна получиться следующая картинка.



3.6 Очистка проекта

Внесем в проект еще несколько изменений. Сейчас у нас можно изменять значение начального преобразования `initT` с помощью нажатия клавиши **М** или изменения размера окна. Отменим такую возможность.

Прежде всего, удалим реакции на клавиши **М** и **Н** (реакция на **Н** стала бесполезной, так как теперь у нас выводится лишь то одно изображение, элементы которого сохранены в списке `figure`).

Далее условимся, что принцип, введенный в разделе 2.2:

1. какой бы ни был размер окна, заданное изображение должно быть максимально возможно увеличено так, чтобы полностью помещаться в окне;
2. точка левого верхнего угла левой верхней клетки заданного изображения должна совпадать с левой верхней угловой точкой.

будет действовать только при первой отрисовке изображения, но не будет действовать при изменении размеров окна при выведенном на него изображении.

Это можно сделать, если вычисление матрицы `initT` производить в момент загрузки изображения.

В обработчике события `Paint` за вычисление `initT` отвечает блок

```
float Wx = ClientRectangle.Width; // размер окна по горизонтали
float Wy = ClientRectangle.Height; // размер окна по вертикали
float aspectForm = Wx / Wy; // соотношение сторон окна рисования
float Sx, Sy;
if (keepAspectRatio) {
    // коэффициенты увеличения при сохранении исходного соотношения сторон
    Sx = Sy = aspectFig < aspectForm ? Wy / Vy : Wx / Vx;
}
else {
    Sx = Wx / Vx; // коэффициент увеличения по оси Ox
    Sy = Wy / Vy; // коэффициент увеличения по оси Oy
}
float Ty = Sy * Vy; // смещение в положительную сторону по оси Oy после смены знака
initT = translate(0.f, Ty) * scale(Sx, -Sy); // преобразования применяются справа налево
// сначала масштабирование, а потом перенос
// в initT совмещаем эти два преобразования
```

Так как мы отменили изменение переменной `keepAspectRatio`, то блок `else` можно выкинуть. Останется фрагмент:

```
float Wx = ClientRectangle.Width; // размер окна по горизонтали
float Wy = ClientRectangle.Height; // размер окна по вертикали
float aspectForm = Wx / Wy; // соотношение сторон окна рисования
float Sx, Sy;
// коэффициенты увеличения при сохранении исходного соотношения сторон
Sx = Sy = aspectFig < aspectForm ? Wy / Vy : Wx / Vx;
float Ty = Sy * Vy; // смещение в положительную сторону по оси Oy после смены знака
initT = translate(0.f, Ty) * scale(Sx, -Sy); // преобразования применяются справа налево
// сначала масштабирование, а потом перенос
// в initT совмещаем эти два преобразования
```

Теперь отпала нужда в двух параметрах масштабирования Sx и Sy . Заменим их одним параметром S . Получим

```
float Wx = ClientRectangle.Width; // размер окна по горизонтали
float Wy = ClientRectangle.Height; // размер окна по вертикали
float aspectForm = Wx / Wy; // соотношение сторон окна рисования
// коэффициент увеличения при сохранении исходного соотношения сторон
float S = aspectFig < aspectForm ? Wy / Vy : Wx / Vx;
float Ty = S * Vy; // смещение в положительную сторону по оси Oy после смены знака
initT = translate(0.f, Ty) * scale(S, -S); // преобразования применяются справа налево
// сначала масштабирование, а потом перенос
// в initT совмещаем эти два преобразования
```

И теперь этот блок перенесем в процедуру `btnOpen_Click` после строки, в которой перевычисляется значение переменной `aspectForm` (строка 31 листинга процедуры `btnOpen_Click` в предыдущем разделе):

```
if (cmd == "frame") { // размеры изображения
    s >> Vx >> Vy; // считываем глобальные значения Vx и Vy
    aspectFig = Vx / Vy; // обновление соотношения сторон
    float Wx = ClientRectangle.Width; // размер окна по горизонтали
    float Wy = ClientRectangle.Height; // размер окна по вертикали
    float aspectForm = Wx / Wy; // соотношение сторон окна рисования
    // коэффициент увеличения при сохранении исходного соотношения сторон
    float S = aspectFig < aspectForm ? Wy / Vy : Wx / Vx;
    float Ty = S * Vy; // смещение в положительную сторону по оси Oy после смены знака
    initT = translate(0.f, Ty) * scale(S, -S); // преобразования применяются справа налево
    // сначала масштабирование, а потом перенос
    // в initT совмещаем эти два преобразования
}
```

Наконец, так как матрица `initT` после загрузки изображения из файла перевычисляться не будет, то отпадает необходимость в матрице M : можно просто матрице T всегда, в качестве начального значения, присваивать матрицу `initT`. Первый раз сделаем это сразу в процедуре `btnOpen_Click`, после вычисления `initT`.

```
T = initT;
```

То же самое нужно будет сделать в обработчике события `KeyDown` при нажатии клавиши **Escape** (вместо того, чтобы присваивать единичную матрицу).

Таким образом, матрицу M в обработчике события `Paint` заменим на T .

После этих изменений процедура `MyForm_Paint` примет вид

```
1 private: System::Void MyForm_Paint(System::Object^ sender, System::Windows::Forms::PaintEventArgs^ e) {
2     Graphics^ g = e->Graphics;
3     g->Clear(Color::AquaMarine);
4
5     for (int i = 0; i < figure.size(); i++) {
```

```

6 path lines = figure[i]; // lines - очередная ломаная линия
7 Pen pen = gcnew Pen(Color::FromArgb(lines.color.x, lines.color.y, lines.color.z));
8 pen->Width = lines.thickness;
9
10 vec2 start = normalize(T * vec3(lines.vertices[0], 1.0)); // начальная точка первого отрезка
11 for (int j = 1; j < lines.vertices.size(); j++) { // цикл по конечным точкам (от единицы)
12     vec2 end = normalize(T * vec3(lines.vertices[j], 1.0)); // конечная точка
13     g->DrawLine(pen, start.x, start.y, end.x, end.y); // отрисовка отрезка
14     start = end; // конечная точка текущего отрезка становится начальной точкой следующего
15 }
16 }
17 }

```

Процедура MyForm_KeyDown примет вид

```

1 private: System::Void MyForm_KeyDown(System::Object^ sender, System::Windows::Forms::KeyEventArgs^ e) {
2     float Wcx = ClientRectangle.Width / 2.f; // координаты центра
3     float Wcy = ClientRectangle.Height / 2.f; // текущего окна
4     switch (e->KeyCode) {
5     case Keys::Escape:
6         T = initT;
7         break;
8     case Keys::Q:
9         T = translate(-Wcx, -Wcy) * T; // перенос начала координат в (Wcx, Wcy)
10        T = rotate(0.01f) * T; // поворот на 0.01 радиан относительно
11            // нового центра
12        T = translate(Wcx, Wcy) * T; // перенос начала координат обратно
13        break;
14    case Keys::W:
15        T = translate(0.f, -1.f) * T; // сдвиг вверх на один пиксел
16        break;
17    default:
18        break;
19    }
20    Refresh();
21 }

```

Наконец, процедура btnOpen_Click примет вид

```

1 private: System::Void btnOpen_Click(System::Object^ sender, System::EventArgs^ e) {
2     if (openFileDialog->ShowDialog() == System::Windows::Forms::DialogResult::OK) {
3         // в файловом диалоге нажата кнопка OK
4         // перезаписи имени файла из openFileDialog->FileName в fileName
5         wchar_t fileName[1024]; // переменная, в которой посимвольно сохраним имя файла
6         for (int i = 0; i < openFileDialog->FileName->Length; i++)
7             fileName[i] = openFileDialog->FileName[i];
8         fileName[openFileDialog->FileName->Length] = '\0';
9
10        // объявление и открытие файла
11        ifstream in;
12        in.open(fileName);
13        if (in.is_open()) {
14            // файл успешно открыт
15            figure.clear(); // очищаем имеющийся список ломаных
16            // временные переменные для чтения из файла
17            float thickness = 2; // толщина со значением по умолчанию 2
18            float r, g, b; // составляющие цвета
19            r = g = b = 0; // значение составляющих цвета по умолчанию (черный)
20            string cmd; // строка для считывания имени команды
21            // непосредственно работа с файлом
22            string str; // строка, в которую считываем строки файла
23            getline(in, str); // считываем из входного файла первую строку
24            while (in) { // если очередная строка считана успешно
25                // обрабатываем строку
26                if ((str.find_first_not_of(" \t\r\n") != string::npos) && (str[0] != '#')) {
27                    // прочитанная строка не пуста и не комментарий
28                    stringstream s(str); // строковый поток из строки str
29                    s >> cmd;
30                    if (cmd == "frame") { // размеры изображения
31                        s >> Vx >> Vy; // считываем глобальные значения Vx и Vy
32                        aspectFig = Vx / Vy; // обновление соотношения сторон
33                        float Wx = ClientRectangle.Width; // размер окна по горизонтали
34                        float Wy = ClientRectangle.Height; // размер окна по вертикали
35                        float aspectForm = Wx / Wy; // соотношение сторон окна рисования
36                        // коэффициент увеличения при сохранении исходного соотношения сторон
37                        float S = aspectFig < aspectForm ? Wy / Vy : Wx / Vx;
38                        float Ty = S * Vy; // смещение в положительную сторону по оси Oy после смены знака
39                        initT = translate(0.f, Ty) * scale(S, -S); // преобразования применяются справа налево
40                        // сначала масштабирование, а потом перенос
41                        // в initT совмещаем эти два преобразования
42
43                        T = initT;
44                    }
45                    else if (cmd == "color") { // цвет линии
46                        s >> r >> g >> b; // считываем три составляющие цвета
47                    }
48                    else if (cmd == "thickness") { // толщина линии
49                        s >> thickness; // считываем значение толщины
50                    }

```

```

50     else if (cmd == "path") { // набор точек
51         vector<vec2> vertices; // список точек ломаной
52         int N; // количество точек
53         s >> N;
54         string str1; // дополнительная строка для чтения из файла
55         while (N > 0) { // пока не все точки считали
56             getline(in, str1); // считываем в str1 из входного файла очередную строку
57             // так как файл корректный, то на конец файла проверять не нужно
58             if ((str1.find_first_not_of(" \t\r\n") != string::npos) && (str1[0] != '#')) {
59                 // прочитанная строка не пуста и не комментарий
60                 // значит в ней пара координат
61                 float x, y; // переменные для считывания
62                 stringstream s1(str1); // еще один строковый поток из строки str1
63                 s1 >> x >> y;
64                 vertices.push_back(vec2(x, y)); // добавляем точку в список
65                 N--; // уменьшаем счетчик после успешного считывания точки
66             }
67         }
68         // все точки считаны, генерируем ломаную (path) и кладем ее в список figure
69         figure.push_back(path(vertices, vec3(r, g, b), thickness));
70     }
71 }
72 // считываем очередную строку
73 getline(in, str);
74 }
75 Refresh();
76 }
77 }
78 }

```

Удалим из проекта те инструкции, которые нам больше не нужны.

Прежде всего, в блоке описания глобальных переменных оставим лишь следующие объявления.

```

1 float Vx; // размер рисунка по горизонтали
2 float Vy; // размер рисунка по вертикали
3 float aspectFig; // соотношение сторон рисунка
4 vector<path> figure;
5 mat3 T; // матрица, в которой накапливаются все преобразования
6 mat3 initT; // матрица начального преобразования

```

Параметр `keepAspectRatio` стал не нужен. Удалим его объявление. То же самое касается возможных параметров, которые Вы добавляли в приложении при выполнении задания 2.

И теперь стало ненужным все, что содержится в обработчике события *Load*. Оставим этот метод пустым.

3.7 Задание для самостоятельной работы

Задание 3

1. Создайте приложение, включающее в себя все, что было описано выше в качестве примера. Проект должен называться Вашей фамилией, записанной латинскими буквами.
2. В заголовочном файле `Transform.h` определите процедуры `mirrorX` и `mirrorY` без параметров, выдающие матрицы зеркального отражения относительно осей *Ox* и *Oy* соответственно.
3. Назначьте реакции на нажатие клавиш
 - **E** — поворот изображения по часовой стрелке на 0.01 радиан относительно текущего центра окна;
 - **S, A, D** — сдвиг изображения соответственно вниз, влево и вправо на 1 пиксел;
 - **R, Y** — поворот изображения соответственно по и против часовой стрелки на 0.05 радиан относительно текущего центра окна;

- **T,G,F,H** — сдвиг изображения соответственно вверх, вниз, влево и вправо на 10 пикселей;
 - **Z,X** — соответственно увеличение и уменьшение изображения относительно текущего центра окна в 1.1 раза;
 - **U,J** — зеркальное отражение относительно горизонтальной и вертикальной оси, проходящей через центр окна, соответственно;
 - **I,K** — соответственно растяжение и сжатие изображения по горизонтали относительно текущего центра окна в 1.1 раза;
 - **O,L** — соответственно растяжение и сжатие изображения по вертикали относительно текущего центра окна в 1.1 раза;
4. Создайте текстовый файл, описывающий изображение, соответствующее Вашему варианту в задании 2, который корректно загружается и отображается в проекте. Разбейте Ваше изображение на осмысленные фрагменты (посоветуйтесь с преподавателем о том, из каких фрагментов должна быть выполнена Ваша картинка), для которых задайте свои цвета и толщину линий.
 5. Архив получившегося проекта загрузите на портал как ответ на задание 3. Текстовый файл с описанием Вашего изображения должен быть включен в архив.

3.8 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Что такое «однородные координаты» точки?
2. Откуда берется содержимое матриц элементарных преобразований, описанных и реализованных в разделе 3.3.2?
3. Что произойдет, если в преобразованиях умножить вектор на матрицу не слева, а справа?
4. Матрицей какого преобразования является матрица `initT`?
5. Матрицей какого преобразования является матрица `T`?
6. Как организуется поворот изображения относительно начала координат? Из каких элементарных преобразований строится это преобразование?
7. Как организовать зеркальное отражение изображения относительно вертикальной оси, проходящей через центр окна?
8. Что произойдет с изображением, если вместо масштабирования относительно центра окна использовать элементарные преобразования масштабирования?
9. Что такое «составляющие RGB»?
10. Как можно получить цвет объекта Windows Forms из составляющих RGB?
11. Откуда в результирующем проекте берутся цвета линий изображения?