КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Учебно-методическое пособие по курсу

Компьютерный практикум (часть 2)

С.В. МАКЛЕЦОВ

Казань — 2013

УДК 004.43

Печатается по решению Редакционно-издательского совета ФГАОУВПО «Казанский (Приволжский) федеральный университет»

методической комиссии Института математики и механики им. Н.И. Лобачевского
Протокол № 2 от 15 ноября 2012 г.

заседания кафедры теории функций и приближений
Протокол № 3 от 2 ноября 2012 г.

Научный редактор —
доктор физ.-мат. наук, проф. Ф.Г. Авхадиев

Рецензенты:
канд. физ.-мат. наук, доц. Е.К. Липачев,
канд. физ.-мат. наук, доц. Д.Н. Тумаков

Маклецов С.В.

Компьютерный практикум (часть 2). Учебно-методическое пособие. — Казань: Казанский (Приволжский) федеральный университет, 2013. — 101 с.

В методической разработке приведена программа практических занятий по 2 части курса «Компьютерный практикум» для бакалавров 3 курса института математики и механики им. Н.И. Лобачевского Казанского (Приволжского) федерального университета. Пособие также включает теоретические сведения по изучаемым темам, разобранные примеры программ, написанных на языке программирования C++/CLI, задания для самостоятельной работы, список рекомендуемой литературы.

© Казанский университет, 2013
© Маклецов С.В., 2013

Оглавление

Введение 5
Рабочая программа 2 части курса «Компьютерный практикум». 6
I. Расширение языка C++/CLI для создания управляемого кода 8
А. Теоретическая справка 8
1. Неуправляемые классы — class 11
2. Ссылочные классы — ref class 12
3. Классы-значения — value class 14
4. Структуры-ссылки и структуры-значения — ref struct и value struct 14
5. Классы-перечисления — enum class 15
6. Интерфейсные классы — interface class 15
7. Управляемые массивы — array 17
8. Свойства—property 19
Б. Пример консольной программы на языке C++/CLI 21
В. Задания для самостоятельного выполнения 26
II. Создание приложений, использующих графический интерфейс пользователя (Windows Forms). 27
А. Теоретическая справка 27
1. Использование графических элементов управления 27
2. Динамическое формирование изображений. 30
3. Обобщенные типы— generic 41
4. Коллекции—Collections 41
Б. Примеры 45
В. Задания для самостоятельного выполнения 59
III. Создание приложений с несколькими потоками. Синхронизация потоков в C++/CLI. 63
А. Теоретическая справка 63
1. Создание и использование потоков. 63

2. Синхронизация потоков. 66
3. Делегаты. Модель событий. 68
Б. Примеры 69
В. Задания для самостоятельного выполнения 90
IV. Использование параллельных вычислений для решения математических задач с использованием классов библиотеки .NET 91
А. Теоретическая справка 91
Б. Примеры 93
В. Задания для самостоятельного выполнения 98
Литература 100

Введение

Уважаемые студенты! В рамках второй части курса «Компьютерный практикум», вам предлагается изучить язык программирования C++/CLI, являющийся расширением языка C++ для платформы .NET от Microsoft.

Для успешного освоения материала и приобретения практики программирования вам потребуется установить среду разработки MS Visual Studio версии 2012 (Professional) или выше.

Настоящее пособие содержит учебную программу по второй части курса «Компьютерный практикум», а также теоретический материал, примеры программ и задания для самостоятельного выполнения для каждой из тем учебной программы.

В рамках второй части курса изучаются следующие темы:

- Расширение языка C++/CLI для создания управляемого кода .NET;
- Создание приложений, использующих графический интерфейс пользователя (Windows Forms);
- Создание приложений с несколькими потоками; синхронизация потоков в C++/CLI;
- Использование параллельных вычислений для численного решения математических задач с использованием классов библиотеки .NET.

Рабочая программа 2 части курса «Компьютерный практикум».

Вторая часть курса «Компьютерный практикум», входящего в учебный план направления 010100 — «Математика», в соответствии с учебной программой содержит четыре темы.

Тема 11.

Расширение языка C++/CLI. Особенности работы с реализацией языка C++ для платформы .NET. Управляемые объекты. Отличие от языка C++.

В рамках темы изучаются:

- примитивные типы языка C++/CLI;
- ссылочные типы, в том числе:
  - ссылочные классы и классы-значений, их отличие от неуправляемых классов;
  - структуры-ссылки и структуры-значения;
- классы-перечисления;
- интерфейсные классы;
- работа с управляемыми массивами;
- свойство, как элемент класса.

На изучение темы 11 дается 8 часов лабораторных занятий и 8 часов самостоятельной работы.

Тема 12.

Работа с графическим интерфейсом пользователя с помощью проектов Windows Forms на языке C++/CLI.

В рамках темы изучаются:

- способы создания приложений с графическим интерфейсом в режиме визуального программирования;
- стандартные графические элементы управления;
- класс Graphics пространства имен System::Drawing;
- коллекции;
- цикл for each;
- решение задачи теории приближения функций, средствами приложений, разработанных для среды .NET.

На изучение темы 12 дается 12 часов лабораторных занятий и 14 часов самостоятельной работы.

Тема 13.

Создание приложений, допускающих одновременное выполнение нескольких действий. Использование потоков.

В рамках темы изучаются:

- организация работы нескольких потоков в приложении .NET при помощи классов Thread, ThreadStart, ParameterizedThread-Start пространства имен System::Threading;
- средства синхронизации потоков в C++/CLI (класс Monitor);
- делегаты;
- использование модели событий.

Для изучения темы 13 требуется 14 часов лабораторных работ и 14 часов самостоятельной работы.

Тема 14.

Создание приложений, использующих параллельные вычисления при помощи методов платформы .NET.

В рамках темы изучаются:

- использование параллельных вычислений для численного решения математических задач с использованием классов библиотеки .NET.

Для изучения темы 14 необходимо 8 часов лабораторных работ и 10 часов самостоятельной работы.

I. Расширение языка C++/CLI для создания управляемого кода

А. Теоретическая справка

Стандарт языка C++/CLI, предложенный Microsoft, является расширением языка программирования C++ и используется для обеспечения связи между нативными («родными») конструкциями и объектами языка C++ и общей языковой инфраструктурой (CLI – Common Language Infrastructure) платформы .NET. Целью создания языка C++/CLI были:

- предоставление удобного и универсального синтаксиса и семантики, знакомого программистам, использующим язык C++;
- обеспечение поддержки для ряда особенностей инфраструктуры CLI платформы .NET (свойств, событий, сборки мусора, обобщений) для всех типов, включая стандартные классы языка C++;
- обеспечение поддержки особенностей стандартного языка C++, для обеспечения совместимости с большим количеством ранее написанного кода.

Примечание. *При программировании на языке C++/CLI в среде MS Visual Studio 2010 отсутствует поддержка IntelliSense. Это значит, что указанная среда разработки не предоставляет пользователю подсказок по написанию кода, например, не выдает список методов для объектов классов, после набора их имени и символа «.», что в ряде случаев, особенно при обучении, существенно усложняет процесс программирования. Поэтому рекомендуется использовать среду MS Visual Studio 2012, в которой работа функции IntelliSense восстановлена.*

Рассмотрим некоторые важные изменения языка C++/CLI по сравнению с языком программирования C++.

а) Примитивные типы.

В рамках программной платформы .NET определена новая, так называемая, *общая система типов* (Common Type System — CTS). Поэтому в задачи языка C++/CLI, входит, в том числе, обеспечение отображения его собственной системы типов на CTS. Ниже, в таблице I-1 показано такое отображение для C++/CLI.

ТАБЛИЦА I-1

Тип в C++	Тип в C++/CLI	Примеры объявления	Описание
bool	**Boolean**	bool flag = true; Boolean flagC = true;	Однобайтовая переменная, которая может содержать одно из двух возможных значений: **`true`** или **`false`**.
char	**SByte**	unsigned char c = 's'; Byte cC = 't';	8-битное целое число со знаком (может быть представлено символом)
double	**Double**	double pi = 3.14159265358979323; Double eC = 2.71828459045	8-байтовое число с плавающей точкой удвоенной точности
float	**Single**	float f = 5.5; Single fC = 4.5;	Вещественное 4-байтовое число.
int, *long*	**Int32**	int i = -10; Int32 k = -90;	32-битное целое число со знаком
long long, *__int64*	**Int64**	__int64 L = -20000000; Int64 LC = 2039485749;	64-битное целое число со знаком
short	**Int16**	short s = -333; Int16 sC = -333;	16-битное целое число со знаком.
unsigned __int64, *unsigned long long*	**UInt64**	unsigned long long k = 9090; UInt64 kC = 9090;	64-битное целое число без знака.
unsigned char	**Byte**	unsigned char c = ‘s’; Byte cC = ‘t’;	8-битное целое число без знака (может быть представлено символом)

Тип в C++	Тип в C++/CLI	Примеры объявления	Описание
unsigned int, unsigned long	**UInt32**	unsigned int i = 10; UInt32 k = 90;	32-битное целое число без знака
unsigned short	**UInt16**	unsigned short us = 15; UInt16 usC = 15;	16-битное целое число без знака.
void	**Void**	Void f(double x);	Тип используется при описании методов для указания того, что возвращаемое значение отсутствует. Также возможно его использование для создания нетипизированных указателей.
wchar_t	**Char**	wchar_t uc = ‘П’; Char ucC = ‘P’;	Символ в кодировке UNICODE

Указанные во втором столбце таблицы типы определены в пространстве имен **`System`**, поэтому для их использования нужно либо явно указывать принадлежность к пространству имен, например, **`System::Int32`**, либо использовать инструкцию "**`using namespace System;`**".

Все типы, относящиеся к CTF, также называются управляемыми, в отличие от «нативных», неуправляемых C++ типов. К последним можно отнести, например, указатели вида int* a.

б) Составные типы.

В таблице I-2 представлены базовые составные типы языка C++/CLI.

ТАБЛИЦА I-2

Название	**Описание**
ref class	Ссылочный класс
ref struct	Ссылочная структура
value class	Класс значений
value struct	Структура значений
enum class	Перечисление
interface class	Интерфейс
array	Управляемые массивы
generic	Обобщенные типы

Помимо того, стоит упомянуть еще о трех типах, которые, хоть и не являются основными, однако, применяются в программах достаточно часто:

- **`System::Object`** — базовый тип для всех управляемых типов, как ссылочных, так и типов-значений;
- **`System::String`** — класс для работы со строками, хранящимися в кодировке Unicode, и предоставляющий методы для работы с ними;
- **`System::Decimal`** — класс, для работы с десятичными числами с не менее чем 28 значащими цифрами.

Итак, типы библиотеки CTS, используемые в языке C++/CLI, являются управляемыми. В частности, это означает, что созданные в памяти объекты не требуют удаления: за программиста это делает сборщик мусора, встроенный в исполняющую среду .NET Framework.

Рассмотрим составные типы подробнее.

1. Неуправляемые классы — class

Классы, описанные стандартным образом, являются неуправляемыми. Пример описания неуправляемого класса приведен ниже.

```
class NClass {
private:
    int v;
public:
    NClass(void) {//Конструктор по умолчанию
        v = 0;
    }
    void SetV(int v){//Установить значение поля v
        //Значение должно быть неотрицательным
        this->v = (v>0)?v:0;
    }

    int GetV() {//Получить значение поля v
        return v;
    }
};
```

Такие классы поддерживаются языком C++/CLI для обеспечения совместимости со стандартным C++, но они не отображаются на библиотеку типов CTS.

Следующий код содержит допустимые действия над объектами описанного выше класса.

```
NClass nc; //Создание объекта класса
NClass *pnc = new NClass();//Создание указателя на объект класса.
nc.SetV(15);
cout << pnc->GetV(); // 0
pnc = &nc;
cout << pnc->GetV(); // 15
```

2. Ссылочные классы — ref class

Для введения управляемых классов их нужно объявлять либо как ссылочные, либо как классы-значения.

Основное отличие классов значений от ссылочных классов состоит в том, что переменные первых хранят непосредственно сами значения, а вторых — ссылки на значения. При использовании ссылочных классов имеется возможность создавать разные переменные, указывающие на одни и те же объекты в памяти. Поэтому при модификации данных в любой такой переменной, изменения будут происходить с общим объектом и, стало быть, проявятся при работе с каждой из переменных. Напротив, при использовании классов-значений каждая пе-

ременная имеет собственную копию объекта в отдельной области памяти, а значит операции, производимые над одним объектом, не будут влиять на другой.

Приведем пример работы с управляемым классом.

```
//Для поддержки названий типов библиотеки CTS необходимо
//подключить системное пространство имен:
using namespace System;
ref class RClass {//Описание ссылочного класса
private: Int32 v;
public:
    RClass(void){//Конструктор по умолчанию
        v = 0;
    }
    Void SetV(Int32 v)   {
        this->v = v;
    }
    Int32 GetV() {
        return v;
    }
};
```

Следующий код содержит примеры допустимых действий над объектами ссылочного класса.

```
RClass^ prc = gcnew RClass();
RClass^ prc2 = prc;
prc->SetV(20);
cout << prc2->GetV() << endl;//20
```

Обратите внимание на изменения в синтаксисе языка. Так, для создания управляемого указателя применяется знак «**^**», вместо «*****», который продолжает обозначать неуправляемый указатель, применяемый в стандартном C++. Ключевое слово **new**, используемое в нативном C++ для выделения памяти, заменено для управляемых объектов на **gcnew**. Несложно догадаться, что приставка «gc», происходит от акронима GC — Garbage Collection (сборка мусора) и говорит об управляемости такого указателя. Ключевое слово **delete** в данном случае можно не использовать, поскольку программисту не требуется очищать память.

3. *Классы-значения — value class*

```
//Пример описания класса-значения.
using namespace System;
value class VClass {//Класс-значение
private:
    Int32 v;
public:
    //Конструкторы по умолчанию в классах-значениях запрещены,
    //поэтому используется конструктор с параметром
    VClass(int val){
        v = val;
    }
    Void SetV(Int32 v)   {
        this->v = v;
    }
    Int32 GetV() {
        return v;
    }
};
```

В классах-значениях запрещается создавать конструктор по умолчанию, поскольку он уже неявно добавляется исполняющей средой CLR во все типы-значения. Все переменные value-классов, объекты которых созданы при помощи неявных конструкторов по умолчанию, инициализируются нулями.

Пример кода, содержащего допустимые действия над объектами класса-значения:

```
VClass vc(10);
VClass vc2;
Console::WriteLine(vc.GetV());// 10
Console::WriteLine(vc2.GetV());// 0
```

4. *Структуры-ссылки и структуры-значения — ref struct и value struct*

Создание управляемых структур в C++/CLI (**ref struct** и **value struct**) по сути ничем не отличается от классов. Единственным отличием этих объектов является то, что в структурах, поля́, для которых явно не задан модификатор приватности, являются общедоступными (**public**), в то время как у классов, поля по умолчанию защищены (**private**).

5. Классы-перечисления — enum class

Класс-перечисление пришел в C++/CLI на замену обычным перечислениям (enum) языка C++. Класс-перечисление имеет простую структуру и определяет ряд именованных значений. При этом он фактически является value-типом. Пример создания enum-класса приведен ниже.

```
public enum class Month{
    January = 1, February, March, April, May, June,
July, August, September, October, November, December
    /**Класс содержит перечисление месяцев: Январь
    (этому элементу присваивается порядковый номер 1),
     Февраль (2), Март (3), Апрель (4) и т.д.*/
};
```

Благодаря тому, что класс-перечисление, как и любой другой управляемый тип, наследует от класса **Object**, в состав которого входит метод **ToString**, у программиста есть возможность применить его для получения текстового значения элемента перечисления, что удобно, например, для вывода на экран:

```
void PrintMonthByNum(int i){
    Month m = safe_cast<Month>(i);
    Console::WriteLine(m.ToString());
}
```

6. Интерфейсные классы — interface class

Интерфейсные классы (интерфейсы) отсутствовали как таковые в стандартном C++, хотя, схожей функциональности можно было добиться при использовании абстрактных классов с чисто виртуальными методами. Однако и в этом случае сходство таких классов с интерфейсами не полное. Интерфейсы не могут содержать полей. В них есть только описания методов. Появление интерфейсов связано с тем, что в C++/CLI запрещено множественное наследование для управляемых классов. Любой управляемый класс может наследовать только от одного класса-предка. В качестве компенсации классы могут реализовывать произвольное количество интерфейсов. Класс, реализующий интерфейс, обязан содержать те методы, которые заявлены в интерфейсе.

Имена интерфейсных классов принято начинать с буквы «**I**» (от слова Interface). При этом заканчиваются имена обычно на –**able**, что

говорит о том, что интерфейсы определяют некоторые способности (англ. «able» — уметь, быть способным сделать что-то).

Решение об отказе от множественного наследования довольно спорное. В некоторых случаях, использование таких конструкций вполне логично и упрощает восприятие. Однако в случае большой развернутой структуры классов, использующих механизмы множественного наследования, быстро возникает путаница и трудно определить какие именно методы должны вызываться. Это приводит к ошибкам и непониманию программ.

Пример использования интерфейсного класса приведен ниже:

```
interface class IConvertable
{
    double A2B(double a);
    double B2A(double b);
};
//Класс TempConverter, реализующий интерфейс IConvertable
ref class TempConverter: IConvertable
{
public:
    virtual double A2B(double a){
        // Преобразует температуру из градусов Фаренгейта
        //в градусы Цельсия
        double b = 5./9*(a-32);
        return b;
    }
    virtual double B2A(double b){
        // Преобразует температуру из градусов Цельсия
        // в градусы Фаренгейта
        double a = 9./5*b+32;
        return a;
    }
};
//Класс MoneyConverter, реализующий интерфейс IConvertable
ref class MoneyConverter: IConvertable
{
    //Осуществляет преобразование между валютами
    //(например, RUR — USD)
public:
    virtual double A2B(double a){
        double b = 30*a;
        return b;
    }
```

```
    virtual double B2A(double b){
        double a = b/30;
        return a;
    }
};
```

7. Управляемые массивы — array

Интересной особенностью управляемых классов является также то, что в них нельзя создавать массивы при помощи стандартного синтаксиса языка C++. Например, такой код вызовет ошибку компиляции:

```
ref class RClass {  //Управляемый класс
private:
    Int32 v;
    double d[10];  /*Ошибка: член управляемого класса не может
                        быть неуправляемым массивом */
};
```

Для создания управляемых массивов в C++/CLI используется новый синтаксис. Так, например, чтобы объявить массив вещественных чисел из 10 элементов, нужно написать:

```
int n = 10;
array<double>^ d = gcnew array<double>(n);
```

Отсюда видно, что для создания массивов необходимо воспользоваться ключевым словом **array**, после которого в угловых скобках задать тип элементов. Дальнейшая работа с одномерными массивами не будет отличаться от такового в стандартном C++. Скажем, для задания значений элементам массива можно написать:

```
for (int i = 0; i<n; i++){
    d[i] = 1./(i*i+1);
}
```

Работа с многомерными массивами несколько отличается от стандартного способа обозначения элементов, применяемого в нативном C++, и осуществляется следующим образом:

```
int m = 5;//Число строк
int n = 10;//Число столбцов

//Создаем массив размерности 2 размера m*n
array<double, 2>^ matr = gcnew array<double, 2>(m, n);
//...
```

```
//Работа с элементами двумерного массива:
for (int i=0; i<m; i++){
    for (int j=0; j<n; j++){
        matr[i,j] = i*j;
    }
}
```

Еще одним нововведением языка является новый для C++ тип цикла **for each**, который может быть полезен при работе с коллекциями элементов, позволяя пробежаться по всем объектам из некоторой коллекции. (Подробнее о коллекциях будет рассказано в следующей части данного пособия.)

Например, с помощью цикла **for each** можно организовать вывод на экран всех элементов описанного выше одномерного массива **d**:

```
for each (double elem in d)
{
    Console::WriteLine(elem);
}
```

Данный вид цикла удобно применять и для двумерных массивов:

```
int k = 0;//Счетчик элементов
for each (double elem in matr)
{
    /*Форматированный вывод: на каждое число отводится по 2 знака,
     между числами ставится пробел.*/
    Console::Write("{0,2} ", elem);
    if ((++k % n) == 0)
    {
        //После последнего элемента в строке матрицы
        //выводим на экран символ перехода на новую строку
        Console::WriteLine("");
    }
}
```

Из последнего примера видно, что управляемый многомерный массив C++/CLI, в отличие от стандарта языка C++, представляет собой не массив массивов, а массив исходных элементов (в данном случае типа double).

8. Свойства—property

В классах, написанных на языке C++/CLI, наряду с полями и методами, появляются новые члены, занимающие промежуточное положение между ними, — свойства.

С одной стороны, при работе с классом мы можем обратиться к его свойству как к полю, присвоив ему какое-либо значение (set), или наоборот, получив некоторое значение, хранящееся в объекте класса (get).

Например, если в объекте X некоторого класса описано свойство с именем **Count**, мы можем поместить в него значение следующим образом:

```
X.Count = 10;
```

Или, например, получить значение:

```
if (X.Count < 2)
{
    //...
}
```

Такое использование внешне ничем не отличается от работы с общедоступными полями.

С другой стороны, свойства позволяют выполнять определенный набор действий перед заданием или чтением значений, что роднит их с методами.

Предположим, в классе имеется защищенное поле, которое должно содержать целочисленные значения, строго большие 2. Чтобы избежать возможных ошибок программистов, которые будут пользоваться таким классом, необходимо ограничить их непосредственный доступ к такому полю извне. Проверку вводимых значений можно организовать посредством методов. Однако это усложнит синтаксис программы. В этом случае, удобнее использовать именно свойство, содержащее стандартные методы **get** и/или **set**. Его описание в классе может быть сделано следующим образом:

```
public ref class CX
{
private:
    int cnt;
public:
    //Определяем свойство
    property int Count
    {
        //метод получения данных
        int get() {
            return cnt;
        }
        //метод задания значения для поля cnt
        void set(int value){
            /*Осуществляем проверку. В случае, когда
            присваиваемое значение окажется меньше 2,
            в поле попадет наименьшее допустимое значение (2). */
            cnt = (value>=2)?value:2;
        }
    }
};
```

Кроме того, при помощи свойств можно открывать избирательный доступ к полю. Например, для того, чтобы сделать свойство **Count** доступным только для записи, можно удалить из него метод **get**. Тогда получить значение извне будет невозможно. Однако останется возможность задавать значения для Count. Напротив, при удалении метода **set** свойство будет доступным только для чтения.

Свойства в C++/CLI могут быть индексированными. Например, если при помощи свойства организуется доступ к элементам массива, возможная конструкция будет выглядеть следующим образом:

```
public ref class CY
{
private:
    short n;//Количество элементов массива
    array<float>^ d; //Массив вещественных чисел
public:
    CY(){
        //Конструктор создает массив
        n = 10;
        d = gcnew array<float>(n);
    }
```

```
    //Свойство, возвращающее и задающее значение элемента массива
    //с некоторым  индексом i
    property float Elem[int]{
        float get(int i){ return d[i];}
        void set (int i, float value){
            if (i<0 || i>=n){
                throw (String^)"Задан неверный индекс";
            } else {
                d[i] = value;
            }
        }
    }
};
```

В данном случае свойство позволяет нам проследить корректность индексации элементов массива, и в случае возникновения ошибки выбросить исключительную ситуацию.

Обратиться к такому свойству в программе можно будет, например, так:

```
CY y;
for (int i = -1; i<= 10; i++){
    try
    {
        y.Elem[i] = (float)i*i;
        Console::WriteLine(i+"): "+y.Elem[i]);
    }
    catch (String^ s)
    {
        Console::WriteLine(s+": "+i);
    }
}
```

В данном случае значениям элементов массива, будут присвоены квадраты их номеров, а на первой и последней итерациях возникнут и будут обработаны исключительные ситуации, с выводом сообщений о неверном индексе элемента.

Б. Пример консольной программы на языке C++/CLI

В данном примере приведена программа, генерирующая и выводящая на экран список равноотстоящих узлов. В программе созданы два класса для работы с узлами:

- **Uzly** – абстрактный класс для описания узлов вообще;
- **RavnootstUzly** — класс-наследник **Uzly** для работы с равноотстоящими узлами.

```
//Файл "Uzly.h"
#pragma once
using namespace System;
ref class Uzly abstract {//абстрактный класс
    //Поля
private:
    int count;//Число узлов
    Decimal left, right;//границы разбиваемого узлами отрезка
protected:
    static Decimal Pi = (Decimal)3.14159265358979232;

    //Методы
public:
    Uzly(int count);
    Uzly(int count, Decimal left, Decimal right);

    //Свойства
public:
    property int Count {//Задание и получение числа узлов
        int get(){
            return count;
        }
        //Должно быть как минимум 2 узла,
        //поэтому делаем проверку:
        void set(int value){
            count = (value < 2) ? 2 : value;
        }
    }
    //Задание и получение левой границы отрезка
    property Decimal Left
    {
        Decimal get(){
            return left;
        }
        void set(Decimal value){
            left = value;
            //Левая граница не должна быть больше правой
            if (right<=left) right = ++left;
        }
    }
```

```
    //Задание и получение правой границы отрезка
    property Decimal Right {
        Decimal get(){
            return right;
        }
        void set(Decimal value){
            right = value;
            //Правая граница не должна быть меньше левой
            if (left>=right) left = --right;
        }
    }
    //Получение узла с номером i при помощи индексированного свойства
    virtual property Decimal Uzel[int]{
        Decimal get(int i) = 0;//абстрактный метод
    }
};
```

```
//Файл "Uzly.cpp"
#include "Uzly.h"
Uzly::Uzly(int count){
    Count = count;
    left = -1;
    right = 1;
}
Uzly::Uzly(int count, Decimal left, Decimal right){
    this->Count = count;
    this->Left = left;
    this->Right = right;
}
```

```
//Файл "RavnootstUzly.h"
#pragma once
#include "uzly.h"
ref class RavnootstUzly
    : public Uzly  //Класс RavnootstUzly наследует от класса Uzly
{
public:
    //Конструкторы
    RavnootstUzly(int count);
    RavnootstUzly(int count,
                  Decimal left,
                  Decimal right);
```

```
    //Переопределенное индексированное свойство
    //Получение узла с номером i (нумерация идет с 0)
    virtual property Decimal Uzel[int]
    {
        virtual Decimal get(int i) override
        {
            if (i<0 || i>=Count){
                throw (String^)"Индекс узла не"\
                                    " определен\n";
            }
            return Left+i*(Right-Left)/(Count-1);
        }
    }
};
```

```
//Файл "RavnootstUzly.cpp"
#include "stdafx.h"
#include "RavnootstUzly.h"
RavnootstUzly::RavnootstUzly(int count)
    : Uzly (count) //Обращение к конструктору базового класса
{
}
RavnootstUzly::RavnootstUzly(int count,
                             Decimal left,
                             Decimal right)
    : Uzly (count, left, right)
{
}
```

```
//Файл "ConsoleCLR.cpp" (главный файл проекта)
#include "Uzly.h"
#include "RavnootstUzly.h"
#include "ChebyshevUzly.h"
using namespace System;
//Метод заполняет массив значениями узлов
array<Decimal>^ FillArray(Uzly^ uzl)
{
    array<Decimal>^ u =
                    gcnew array<Decimal>(uzl->Count);
    for (int i = 0; i<uzl->Count+1; i++){
    //Здесь намеренно допущен выход за границы
    //допустимого диапазона
```

```
        try {
            u[i] = uzl->Uzel[i];
        } catch (String^ s){
            Console::WriteLine("Узел №"+(i)+": "+s);
        }
    }
    return u;
}
//Отображение узлов на экране
void ShowAll(array<Decimal>^ uzl)
{
    for (int i = 0; i<uzl->GetLength(0); i++){
        Console::WriteLine("Узел №"+(i)+"="+uzl[i]);
    }
}
int main(array<System::String ^> ^args)
{
    Console::Write("Укажите количество "\
             "создаваемых узлов: ");
    String^ inp = Console::ReadLine();
    Int16 cnt;
    try {
        //С помощью статического метода Parse класса Int16
        //мы преобразуем введенную строку в число
        cnt = Int16::Parse(inp);
    } catch (...) {
        Console::WriteLine("Неверный ввод! "\
            "Программа не может продолжить работу.");
        return -1;
    }
    //Создаем управляемый указатель на
    //объект «равноотстоящие узлы»
    RavnootstUzly^ uzlR = gcnew RavnootstUzly(cnt);
    //Заполняем узлами массив
    array<Decimal>^ u1 = FillArray(uzlR);
    //Выводим содержимое массива на экран
    Console::WriteLine("\nРавноотстоящие узлы:");
    ShowAll(u1);
    return 0;
}
```

В. Задания для самостоятельного выполнения

1. Дополнить приведенную в примере программу классом **`ChebyshevUzly`** — наследником **`Uzly`** для работы с узлами Чебышева. (Для вычисления узлов Чебышева на отрезке [-1;1] можно воспользоваться формулой: $x_k = \cos\frac{(2k+1)\pi}{2(n+1)}, k = 0,1,\dots,n$. Для вычисления узлов Чебышева на произвольном отрезке [a;b] формула примет вид: $x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\frac{(2k+1)\pi}{2(n+1)}, k = 0,1,\dots,n$).
2. Создать управляемый класс для работы с комплексными числами. Реализовать в классе свойства для задания вещественной и мнимой части числа, а также свойство для задания комплексного числа путем разбора строки вида $x \pm iy$. Реализовать методы сложения и вычитания комплексных чисел. Переопределить операторы «+» и «–». Переопределить метод **`ToString`** для представления комплексного числа в виде строки.
3. В условиях задания 2 реализовать в программе методы для умножения и деления комплексных чисел, возведения комплексного числа в степень.
4. В условиях заданий 2 и 3 создать класс-наследник для работы с комплексными числами в тригонометрической форме записи. Добавить методы вычисления модуля и аргумента комплексного числа.
5. В условиях заданий 2 — 4 создать класс-наследник для работы с комплексными числами в экспоненциальной форме записи.
6. Создать управляемый класс для работы с полиномами. Реализовать в классе свойства и методы, позволяющие: задавать коэффициенты полинома; осуществлять операции сложения, вычитания, умножения полиномов. Переопределить метод **`ToString`** для представления полинома в виде строки.
7. В условиях задания 6 реализовать в классе метод для осуществления операции деления полиномов. Задать индексированное свойство (только для чтения), позволяющее получить значение полинома в указанной точке.
8. В условиях заданий 6 и 7 добавить в класс переопределение оператора индексирования для получения значения указанного коэффициента полинома.

II. Создание приложений, использующих графический интерфейс пользователя (Windows Forms).

А. Теоретическая справка

Язык C++/CLI позволяет создавать приложения Windows Forms, использующие графический интерфейс пользователя, в режиме визуального программирования. Для создания таких приложений можно воспользоваться готовым шаблоном «Windows Forms». После этого в программу можно будет добавлять ряд стандартных графических элементов управления, и формировать внешний вид окна приложения при помощи мышки. Однако в Visual Studio версии 2012 такой шаблон программ для языка C++/CLI отсутствует. Но существует возможность сформировать его из пустого приложения Windows Forms, созданного в Visual Studio 2010 версии, а затем импортировать в Visual Studio 2012. Либо можно создать пустой проект в 2012 версии среды разработки, добавив в него новый класс, созданный по шаблону из группы CLR — «Форма Windows Form».

1. Использование графических элементов управления

При создании приложений с графическим интерфейсом пользователя в среде Visual Studio 2012, у программиста есть возможность пользоваться стандартными графическими элементами управления. При помощи конструктора форм, являющихся заготовками будущих окон программы, существует возможность перетащить мышью элементы, расположенные в палитре, на форму, сформировав тем самым дизайн окна программы. При этом программный код, необходимый для создания экземпляров используемых объектов, будет добавлен в программу автоматически. Такой способ программирования называется визуальным.

Список наиболее распространенных элементов и их назначение приведены в таблице II-1.

ТАБЛИЦА II-1

Вид элемента	Название	Описание
	Button (Кнопка)	Кнопки размещаются на форме, чтобы у пользователя была возможность генерировать определенные события при их нажатии. Таким образом, нажатие кнопок может вызывать выполнение тех или иных действий в программе.
	CheckBox (Переключатель)	Позволяет пользователю выбрать или удалить соответствующий параметр.
	ComboBox (Выпадающий список)	Отображает редактируемое текстовое поле и раскрывающийся список допустимых значений.
	DateTimePicker (Выбор даты)	Позволяет пользователю выбрать дату и время и отобразить их в заданном формате.
	Label (Метка)	Позволяет вывести статический текст, являющийся описанием или информацией, получаемой во время выполнения программы.
	LinkedLabel (Гиперссылка)	Отображает элемент «метка» с поддержкой форматирования и функциональностью гиперссылки.
	ListBox (Список)	Отображает список, в котором пользователь может выбирать некоторые элементы.
	ListView (Коллекция элементов)	Отображает список элементов в одном из пяти представлений. Примером его использования может служить правая часть стандартного окна программы «Компьютер» в Windows.
	MaskedTextBox (Поле для ввода с маской)	Символы, вводимые в строку данного элемента, заменяются символом-маской. Используется для ввода паролей.
	MonthCalendar (Календарь)	Представляет собой постоянно отображаемый на форме календарь, показывающий дни некоторого месяца. Позволяет осуществить выбор даты.

Вид элемента	Название	Описание
	NumericUpDown (Переключатель)	Позволяет вводить числовые значения и содержит кнопки для уменьшения/увеличения данного значения.
	PictureBox (Картинка)	Позволяет разместить на форме картинку из файла.
	ProgressBar (Индикатор выполнения)	Полоса, отображающая ход выполнения операции, показывающая соотношение завершенной и оставшейся для выполнения части процесса.
	RadioButton (Кнопка-селектор)	Позволяет осуществить выбор одного из нескольких вариантов.
	RichTextBox (Расширенный текстовый редактор)	Обеспечивает дополнительные возможности ввода и редактирования текста. Поддерживает форматирование символов и абзацев.
	TextBox (Текстовый редактор)	Обеспечивает возможности ввода неформатированного одно- или многострочного текста.
	ToolTip (Подсказка)	Элемент появляется в виде небольшой подсказки при наведении курсора мыши на другой элемент формы.
	TreeView (Коллекция элементов в форме дерева)	Позволяет отобразить коллекцию связанных друг с другом элементов в форме дерева. Пример — левая часть окна программы «Компьютер» в ОС Windows.
	WebBrowser (Браузер)	Позволяет отображать веб-страницы внутри формы.
	GroupBox (Группирующая панель)	Контейнер, содержащий название в левой верхней части, позволяющий сгруппировать некоторые элементы формы по смыслу.
	Panel (Панель)	Контейнер, позволяющий осуществить группировку элементов по смыслу или отобразить динамически сформированное содержимое.
	TabControl (Панель вкладок)	Обрабатывает и отображает для пользователя коллекцию вкладок, между которыми можно осуществлять переключение.

Вид элемента	Название	Описание
	TrackBar (Ползунок)	Позволяет пользователю выбрать значение из некоторого диапазона, перемещая ползунок по полосе.
,	VScrollBar, HScroll-Bar (Полосы прокрутки)	Используются для вертикальной и горизонтальной прокрутки элементов, не умещающихся в пределах отображаемой области.
	MenuStrip (Строка меню)	Полоса меню, расположенная в верхней части окна приложения.
	StatusStrip (Строка статуса)	Располагается в нижней части окна. Показывает пользователю информацию о просматриваемом объекте, компонентах объекта или производимых операциях.
	Chart (Диаграмма)	Позволяет отображать на форме различные виды диаграмм и графиков.

Помимо того, что было перечислено в этой таблице, палитра Visual Studio содержит и другие графические компоненты, позволяющие организовать различную функциональность программ: работу с диалоговыми окнами открытия/сохранения файла, диалогом выбора цвета; организацию печати и предпросмотра документа перед печатью; работу с процессами и таймером; доступ к данным из хранилищ данных (баз данных) и т.д.

2. Динамическое формирование изображений.

Для рисования в оконном компоненте программы с графическим интерфейсом посредством языка C++/CLI можно воспользоваться классами, входящими в пространство имен **`System::Drawing`**, которое обеспечивает доступ к функциональным возможностям графического интерфейса GDI+. Пространства имен **`System::Drawing::Drawing2D`**, **`System::Drawing::Imaging`**, и **`System::Drawing::Text`** обеспечивают дополнительные функциональные возможности.

Основным классом, который используется для динамического формирования изображения на некотором устройстве отображения, является **`Graphics`**, который инкапсулирует поверхность рисования GDI+, предоставляя методы для формирования изображений. Этот

класс не наследуется и предоставляет методы для вывода объектов в устройстве отображения. Объект класса **Graphics** связан с конкретным контекстом устройства.

Его можно получить для любого отображаемого на экране объекта, наследующего от **System::Windows::Forms::Control**, путем вызова метода **Control::CreateGraphics** или путем обработки события **Control::Paint** элемента управления. Во втором случае необходимо обратиться к свойству **Graphics** класса **PaintEventArgs** пространства имен **System::Windows::Forms**, объект которого передается в метод в качестве параметра. Можно также создать объект **Graphics** из изображения, используя метод **FromImage**.

В таблице II-2 приведены некоторые, наиболее употребимые, элементы класса **Graphics**.

ТАБЛИЦА II-2. КЛАСС GRAPHICS

Методы	Описания
void Clear(Color c)	Очищает всю поверхность рисования и выполняет заливку поверхности указанным цветом фона.
void CopyFromScreen(Point src, Point dest, Size sz)	Копирует участок области экрана с левым верхним углом в точке src, на поверхность Graphics, с левым верхнем углом в точке dest, и размером передаваемой области, заданной параметром sz.
void DrawArc(Pen^ pen, int x, int y, int width, int height, int startAngle, int sweepAngle)	Рисует дугу, которая является частью эллипса, заданного парой координат (x, y), шириной и высотой (width, height), Дуга начинается от угла startAngle, откладываемого от оси Ox по часовой стрелке, и имеет размер sweepAngle градусов. Параметры прорисовки задаются ручкой pen.
void DrawClosedCurve(Pen^ pen, array<Point>^ points)	Строит замкнутую кривую, определяемую массивом точек Point. Параметры прорисовки задаются ручкой pen.
void DrawCurve(Pen^ pen, array<Point>^ points)	Строит незамкнутую кривую, определяемую массивом точек Point. Параметры прорисовки задаются ручкой pen.

Методы	Описания
void DrawEllipse(Pen^ pen, int x, int y, int width, int height)	Рисует эллипс, определяемый ограничивающим прямоугольником, заданным с помощью координат для верхнего левого угла прямоугольника (x, y), высоты и ширины (width, height). Параметры прорисовки задаются ручкой pen.
void DrawImage(Image^ image, int x, int y, int width, int height)	Рисует заданный объект Image (картинку) в указанном месте, используя требуемый размер.
void DrawLine(Pen^ pen, int x1, int y1, int x2, int y2)	Проводит линию, соединяющую две точки, задаваемые парами координат (x1, y1) и (x2, y2).
void DrawPie(Pen^ pen, int x, int y, int width, int height, int startAngle, int sweepAngle)	Рисует сектор, определяемый эллипсом, который задан парой координат (x, y), шириной (width), высотой (height), и двумя радиальными линиями. Первая отклонена по часовой стрелке от оси x на startAngle градусов, вторая — на sweepAngle градусов по часовой стрелке от первой. Параметры прорисовки задаются ручкой pen.
void DrawPolygon(Pen^ pen, array<Point>^ points)	Рисует многоугольник, определяемый массивом структур Point. Параметры прорисовки задаются ручкой pen.
void DrawRectangle(Pen^ pen, int x, int y, int width, int height)	Рисует прямоугольник, который определен парой координат (x, y), шириной (width) и высотой (height). Параметры прорисовки задаются ручкой pen.
void DrawString(String^ s, Font^ font, Brush^ brush, float x, float y)	Создает указываемую текстовую строку в заданном месте (x, y) с помощью определяемых объектов Brush (определяет способ заливки шрифта) и Font (начертание шрифта).

Методы	Описания
void FillClosedCurve(Brush^ brush, array<Point>^ points)	Заполняет внутреннюю часть замкнутой кривой, определяемой массивом структур Point. Способ заливки задается кистью brush.
void FillEllipse(Brush^ brush, int x, int y, int width, int height)	Заполняет внутреннюю часть эллипса, определяемого ограничивающим прямоугольником, заданным с помощью пары координат (x, y), ширины (width) и высоты (height). Способ заливки задается кистью brush.
void FillPie(Brush^ brush, int x, int y, int width, int height, int startAngle, int sweepAngle)	Заполняет внутреннюю часть сектора, определяемого эллипсом, который задан парой координат (x, y), шириной (width), высотой (height), и двумя радиальными линиями. Первая отклонена по часовой стрелке от оси x на startAngle градусов, вторая — на sweepAngle градусов по часовой стрелке от первой. Способ заливки задается кистью brush.
void FillPolygon(Brush^ brush, array<Point>^ points)	Заполняет внутреннюю часть многоугольника, определяемого массивом точек, заданных структурами Point. Способ заливки задается кистью brush.
void FillRectangle(Brush^ brush, int x, int y, int width, int height)	Заполняет внутреннюю часть прямоугольника, который определен парой координат (x,y), шириной (width) и высотой (height). Способ заливки задается кистью brush.
static Graphics^ FromHdc(IntPtr hdc)	Создает новый объект Graphics из указанного дескриптора для контекста устройства.
static Graphics^ FromImage(Image^ image)	Создает новый объект Graphics из указанного рисунка Image.
SizeF MeasureString(String^ text, Font^ font)	Измеряет указанную строку в процессе ее создания с помощью заданного объекта Font. Этот метод возвращает структуру SizeF, представляющую размер для строки, которая определяется параметром text в процессе рисования с использованием шрифта font.

Методы	Описания
void ResetTransform()	Сбрасывает матрицу универсального преобразования данного объекта Graphics и делает ее единичной матрицей.
void RotateTransform(float angle)	Применяет заданное вращение (на угол angle) к матрице преобразования данного объекта Graphics.
void ScaleTransform(float sx, float sy)	Применяет указанную операцию масштабирования (задается коэффициентами sx, sy) к матрице преобразования данного объекта Graphics путем ее добавления к матрице преобразования объекта.
void TranslateTransform(float dx, float dy)	Изменяет начало координат координатной системы путем добавления заданного сдвига (dx, dy) к матрице преобразования данного объекта Graphics.

Для управления выводом изображений можно воспользоваться такими классами пространства имен **`System::Drawing`** как **`Pen`** (он используется для рисования линий и кривых) или наследниками абстрактного класса **`Brush`**, которые используются для различного вида заливки замкнутых фигур.

Ниже, в таблице II-3 приводятся некоторые свойства и методы класса **`Pen`**.

ТАБЛИЦА II-3. КЛАСС PEN

Конструкторы	Описания
Pen(Brush^ brush)	Выполняет инициализацию нового экземпляра класса Pen с заданным объектом Brush.
Pen(Color color)	Инициализирует новый экземпляр класса Pen с указанным цветом.
Pen(Brush^ brush, Single width)	Инициализирует новый экземпляр класса Pen с заданными свойствами brush и width.
Pen(Color color, float width)	Инициализирует новый экземпляр класса Pen с указанными свойствами color и width.

Свойства	Описания
property Brush^ Brush	Получает или задает объект Brush, определяющий атрибуты объекта Pen.
property Color Color	Получает или задает цвет объекта Pen.

Свойства	Описания
property DashStyle DashStyle	Задает стиль линии. Возможные значения для DashStyle: *Solid:* Сплошная линия. *Dash:* Линия, состоящая из штрихов. *Dot:* Линия, состоящая из точек. *DashDot:* Штрих-пунктирная линия. *DashDotDot:* Линия, состоящая из повторяющегося шаблона "штрих-две точки".
property LineJoin LineJoin	Получает или задает общий стиль окончаний для двух последовательных линий, нарисованных при помощи объекта Pen. Возможные значения: *Miter:* Угловое соединение со скосом в 45 градусов. В результате получается острый или обрезанный угол в зависимости от того, превышает ли длина среза ограничение по срезу. *Bevel:* Скошенное соединение. В результате получается угол при диагонали. *Round:* Круговое соединение. В результате получается ровная круговая дуга между двумя линиями. *MiterClipped:* Угловое соединение со скосом в 45 градусов. В результате получается острый или срезанный угол в зависимости от того, превышает ли длина среза ограничение по срезу.
property PenType PenType	Задает тип заполнения, который объект Pen использует для рисования линий.

Свойства	Описания
	Возможные значения для PenType: *SolidColor:* Сплошное заполнение. *HatchFill:* Заполнение штриховкой. *TextureFill:* Заполнение с текстурой точечного рисунка. *PathGradient:* Градиентное заполнение контура. *LinearGradient:* Линейное градиентное заполнение.
property float Width	Получает или устанавливает ширину пера Pen, в единицах объекта Graphics, используемого для рисования.

Методы	Описания
virtual Object^ Clone() sealed	Создает точную копию объекта Pen.

Класс **Brush** определяет объекты, которые используются для заливки замкнутых графических фигур, таких как прямоугольники, эллипсы, многоугольники и т.п. Класс **Brush** является абстрактным, поэтому создание его экземпляров не допускается. Для создания кисти можно воспользоваться одним из наследников класса **Brush**, например, **HatchBrush** (прямоугольная кисть со стилем штриховки, основным цветом и цветом фона), **LinearGradientBrush** (кисть, задающая линейный градиент), **SolidBrush** (сплошная одноцветная кисть) и т.п.

В следующих таблицах представлены некоторые свойства и методы ряда классов кистей.

Таблица II-4. Класс HatchBrush

Конструкторы	Описания
HatchBrush(HatchStyle hatchstyle, Color foreColor, Color backColor)	Инициализирует новый экземпляр класса HatchBrush с указанным перечислением HatchStyle, основным цветом (forecolor) и цветом фона (backColor). Приведем некоторые из возможных значений для параметра HatchStyle:

Конструкторы	Описания
	Horizontal: Шаблон горизонтальных линий. *Vertical:* Шаблон вертикальных линий. *ForwardDiagona:* Шаблон линий по диагонали из верхнего левого угла к нижнему правому углу. *BackwardDiagonal:* Шаблон линий по диагонали от правого верхнего угла к левому нижнему углу. *Cross:* Определяет пересекающиеся горизонтальные и вертикальные линии. *DiagonalCross:* Аналогичен Cross с поворотом на 45°. *Percent50:* Задает пятидесятипроцентную штриховку. Отношение цвета переднего плана к цвету фона равно 50:50. *LightDownwardDiagonal:* Определяет диагональные линии, которые отклоняются вправо от верхних точек к нижним точкам и располагаются на 50 процентов ближе друг к другу, чем у объекта ForwardDiagonal, но эти линии не сглажены. *DashedDownwardDiagonal:* Определяет пунктирные диагональные линии с уклоном вправо от верхних точек к нижним точкам. *DashedUpwardDiagonal:* Определяет пунктирные диагональные линии, имеющие уклон влево от верхних точек к нижним точкам. *SmallConfetti:* Определяет штриховку, похожую на конфетти. *ZigZag:* Определяет горизонтальные линии, состоящие из зигзагов.

Конструкторы	Описания
	Wave: Определяет линии, состоящие из тильд (~). *DiagonalBrick:* Определяет штриховку, которая имеет вид слоев кирпичей с уклоном влево сверху вниз. *HorizontalBrick:* Определяет штриховку, выглядящую как горизонтальные слои кирпичей. *DottedDiamond:* Определяет пересекающиеся прямые и обратные диагональные линии, состоящие из точек. *SmallCheckerBoard:* Определяет штриховку, выглядящую как шахматная доска.

Свойства	Описания
property Color BackgroundColor	Получает цвет интервалов между линиями штриховки, нарисованными данным объектом HatchBrush.
property Color ForegroundColor	Получает цвет линий штриховки, нарисованных данным объектом HatchBrush.
property HatchStyle HatchStyle	Получает стиль штриховки для данного объекта HatchBrush. Ряд возможных значений для свойства HatchStyle приведен в описании конструктора данного класса.

Таблица II-5. Класс LinearGradientBrush

Конструкторы	Описания
LinearGradientBrush(RectangleF rect, Color color1, Color color2, float angle)	Создает новый экземпляр класса LinearGradientBrush на основе прямоугольника (rect), начального (color1) и конечного (color2) цветов, а также угла ориентации (angle).

Свойства	Описания
property array<Color>^ LinearColors	Получает или задает начальный и конечный цвета градиента.
property RectangleF Rectangle	Получает прямоугольную область, которая определяет начальную и конечную точки гради-

Свойства	Описания
	ента. Левая сторона прямоугольника определяет начальную точку линейного градиента, а правая — его конечную точку.

Таблица II-6. Класс SolidBrush

Конструкторы	Описания
SolidBrush(Color color)	Создает сплошную кисть указанного цвета color
Свойства	**Описания**
property Color Color	Получает или задает цвет кисти

Для вывода текста при помощи методов класса **Graphics** необходимо использование объектов класса **Font**. Основные способы их создания, а также некоторые свойства данного класса приведены в следующей таблице. Отметим, что все приведенные ниже свойства дают доступ только на чтение.

Таблица II-7. Класс Font

Конструкторы	Описания
Font(String^ familyName, float emSize)	Инициализирует новый шрифт Font, используя указанный размер (emSize) и строковое представление названия семейства шрифтов familyName (например, «Arial»). Если параметр familyName задает неподдерживаемый шрифт или шрифт, не установленный на компьютере, где выполняется приложение, вместо него будет подставлен шрифт по умолчанию (Microsoft Sans Serif).
Font(String^ familyName, float emSize, FontStyle style)	Инициализирует новый шрифт Font, используя указанный размер (emSize) и строковое представление названия семейства шрифтов familyName (например, «Arial»). Если параметр familyName задает шрифт, не установленный на компьютере, где выполняется приложение, вместо него будет подставлен шрифт по умолчанию. Значение параметра style определяет начертание шрифта и может принимать одно из следующих значений:

Конструкторы	Описания
	Regular: Стандартный текст. *Bold:* Текст, выделенный жирным шрифтом. *Italic:* Текст, выделенный курсивом. *Underline:* Текст, выделенный подчеркиванием. *Strikeout:* Текст с линией посредине.

Свойства	Описания
property bool Bold	Возвращает значение, определяющее, является ли данный шрифт полужирным.
property FontFamily^ FontFamily	Получает объект FontFamily, связанный с данным элементом управления Font.
property int Height	Возвращает значение междустрочного интервала данного шрифта. Междустрочный интервал — это расстояние по вертикали между двумя опорными линиями двух соседних строк в тексте, которое включает как пустое пространство между строками, так и высоту самих знаков.
property bool Italic	Получает или задает значение, указывающее, применено ли к данному шрифту курсивное начертание.
property String^ Name	Возвращает имя начертания этого шрифта Font.
property String^ OriginalFontName	Возвращает имя первоначально заданного шрифта. Фактическое и первоначальное имена могут отличаться, поскольку, например, в случае отсутствия указанного шрифта, фактически системой может быть применен его аналог.
property float SizeInPoints	Возвращает размер максимального пробела данного шрифта Font (в пунктах).
property bool Strikeout	Возвращает значение, указывающее, является ли данный шрифт Font перечеркнутым.
property bool Underline	Возвращает значение, показывающее, является ли этот шрифт Font подчеркнутым.

Для создания приложений, приведенных в примерах ниже, нам также дополнительно потребуются знания еще некоторых конструкций языка C++/CLI.

3. Обобщенные типы— generic

Обобщенные типы данных поддерживаются библиотекой типов CTS и в некотором смысле похожи на шаблоны стандартного C++. Обобщенный тип, в отличие от шаблона, осуществляет параметризацию не во время компиляции программы, а во время ее выполнения. Введение обобщенного типа выполняется при помощи ключевого слова **generic**.

Пример создания и использования обобщенного типа:

```
//Объявление обобщенного класса:
generic <typename T>
ref class Vector{
     T Add(T t) {/*… */}
     void Remove (T t) {/*… */}
     //...
};
//...
//Использование обобщенного класса:
Vector <Int32>^ v1 = gcnew Vector<Int32>();
Vector <Double>^ v2 = gcnew Vector<Double>();
v1->Add(40);
v2->Add(5.5);
v1->Remove(40);
//...
```

4. Коллекции—Collections

Коллекции удобно использовать для решения многих задач программирования. Основным назначением коллекций является стандартизация обработки групп объектов в программе. Все коллекции, существующие в .NET Framework, разработаны на основе четко определенных интерфейсов. Как правило, классов стандартной библиотеки, реализующих эти интерфейсы, вполне достаточно для подавляющего большинства задач. Тем не менее, у программиста имеется возможность реализовать и собственную коллекцию.

В среде .NET Framework поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные.

Необобщенные коллекции реализуют ряд основных структур данных, таких как динамический массив, стек, очередь, а также словари, используемые для хранения пар "ключ-значение". Необобщенные коллекции позволяют хранить данные типа **`Object`**, а также любых наследников этого типа. Поскольку все типы являются наследниками **`Object`**, необобщенная коллекция может хранить произвольный объект. Притом, разные элементы такой коллекции могут иметь разные типы. Классы и интерфейсы необобщенных коллекций находятся в пространстве имен **`System::Collections`**. Если в программе необходимо хранить большое количество однотипных элементов, то необобщенные коллекции лучше не применять: работа с ними происходит медленнее, чем с обобщенными, а также возможны ошибки преобразования типа.

Обобщенные коллекции обеспечивают реализацию нескольких стандартных структур данных, включая списки, стеки, очереди и словари, как и необобщенные коллекции. Но, в отличие от последних, обобщенные коллекции являются строго типизированными. Это означает, что они могут хранить только элементы определенного типа данных, которые совместимы по типу с данной коллекцией. Это делает работу более быстрой и безопасной (исключается возможность несовпадения типов, и нет накладных расходов на приведение типов). Обобщенные коллекции объявляются в пространстве имен **`System::Collections::Generic`**.

Специальные коллекции оперируют объектами некоторых конкретных типов или оперируют объектами определенным образом. Специальные коллекции объявляются в пространстве имен **`System::Collections::Specialized`**.

Поразрядная коллекция. К этому типу относится единственная коллекция **`BitArray`**, объявленная в пространстве имен **`System::Collections`**. Она поддерживает поразрядные операции, то есть операции над отдельными двоичными разрядами, например «И», «ИЛИ», «исключающее ИЛИ». Эта коллекция существенно отличается набором доступных действий от остальных типов коллекций.

Параллельные коллекции, определенные в пространстве имен **`System::Collections::Concurrent`**, поддерживают доступ с использованием нескольких потоков.

Основополагающим для всех коллекций является понятие перечислителя, который поддерживается в необобщенных интерфейсах **IEnumerator** и **IEnumerable**, а также в обобщенных интерфейсах **IEnumerator<T>** и **IEnumerable<T>**. Перечислитель обеспечивает возможность поочередного доступа к элементам коллекции. Любая коллекция обязана реализовать хотя бы одну из версий интерфейса **IEnumerable**. В этом случае для поочередного доступа к содержимому коллекции можно воспользоваться циклом **for each**, который был рассмотрен нами в предыдущей части, на странице 18.

С перечислителем связано еще одно средство, называемое итератором. Оно упрощает процесс создания классов коллекций, например, специальных, поочередное обращение к элементам которых организуется в цикле **for each**.

Рассмотрим некоторые классы-коллекции.

ТАБЛИЦА II-8

Необобщенные классы	**Обобщенные классы**	**Описание**
ArrayList	List<T>	Определяет динамический массив, т.е. такой массив, который может при необходимости изменять свой размер.
Hashtable	Dictionary<Tkey, TValue>	Сохраняет пары "ключ-значение", которые собраны в хэш-таблицу, где хэш-значение используется в качестве ключа. Класс HashTable может использоваться в тех случаях, когда необходимо сохранить не только сами элементы, но и их индексы, не являющиеся подряд идущими натуральными числами.
Queue	Queue<T>	Класс для работы с очередью, которая позволяет организовать работу по принципу «первым пришел — первым обслужен». Для добавления и удаления объектов из очереди используются методы Enqueue и Dequeue соответственно. Метод Peek возвращает элемент с головы очереди, оставляя его при этом в коллекции.

Необобщенные классы	Обобщенные классы	Описание
SortedList	SortedList<T>	Эти классы, как и HashTable (Dictionary<T>) поддерживают коллекцию пар «ключ/значение», но используют для внутреннего хранения данных сам ключ, а не хэш-значения. Таким образом, в данном случае мы получаем список элементов, отсортированных по значению ключа, а не по хэш-значению. Элементы коллекции могут быть доступны как по ключу, так и по индексу.
	SortedDictionary <TKey, TValue>	Классы SortedDictionary<TKey, Tvalue> и SortedList<TKey, TValue> имеют схожую функциональность, но различное внутреннее устройство: • SortedList использует меньше памяти, чем SortedDictionary; • SortedDictionary быстрее вставляет и удаляет элементы; • заполненный класс типа SortedList работает быстрее, если при этом не требуется изменение емкости.
Stack	Stack<T>	Классы для работы со стеком. Стек позволяет организовать работу со списком по принципу «последним пришел — первым обслужен». Классы реализуют стек с помощью таких методов, как Push и Pop, для помещения элементов на стек и удаления их с вершины стека. Метод Peek возвращает верхний элемент, оставляя его при этом в стеке.
	HashSet<T>, SortedSet<T>	Классы представлены только в обобщенном формате. Используются для хранения неповторяющихся элементов. При этом в классе SortedSet элементы автоматически упорядочиваются.

Необобщенные классы	Обобщенные классы	Описание
	LinkedList<T>	Класс представляет собой двунаправленный список, задаваемый первым и последним элементами, в котором каждый элемент ссылается на следующий и предыдущий. Преимущества такой коллекции состоят в том, что добавление и удаление элементов «в середину» происходят очень быстро, поскольку отсутствует необходимость сдвигать ранее существующие элементы. Однако имеются и недостатки. Так, доступ к элементам может быть осуществлен исключительно последовательно, начиная с первого или с последнего.

Б. Примеры

В качестве примера ниже приведена программа, реализующая игру «Жизнь». По правилам этой «игры» на клетчатом поле располагаются некоторое количество «живых» клеток (они закрашиваются некоторым цветом). После задания начального состояния поля, производится генерация очередного «поколения» клеток. Для этого применяются следующие правила:

Если клетка «жива» и рядом с ней располагаются еще две или три «живые» клетки, то рассматриваемая клетка остается «живой», в противном случае она «погибает». Если по соседству с пустой клеткой расположено ровно три «живые», то пустая клетка в новом поколении «оживает». При генерации следующих поколений рассматриваются все соседи, в том числе расположенные по диагонали относительно текущей клетки. Таким образом, у каждой клетки всего 8 соседей.

В проекте должны находиться следующие файлы:

- *main.cpp* — точка входа в программу;
- *Form1.h* — содержит класс, описывающий окно программы;
- *GameGeneration.h, GameGeneration.cpp* — реализуют класс GameGeneration;
- *GameField.h, GameField.cpp* — реализуют класс GameField.

```
//main.cpp — точка входа в программу
#include "Form1.h"//Подключение необходимо для работы с формой
using namespace Form1Namespace;
int main(){
    using namespace System;
    using namespace System::Windows::Forms;
    //Подключение визуальных эффектов для окна формы.
    //Работает начиная с Windows XP и выше
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(0);
    //Запуск приложения с графическим интерфейсом.
    Application::Run(gcnew Form1());
}
```

«Поколения» клеток должны каким-то образом генерироваться и храниться в памяти. Для реализации этой задачи создадим класс «**GameGeneration**»:

```
//Файл «GameGeneration.h»
#pragma once
namespace GameLife{
    ref class GameGeneration {
        //Поля класса:
    private:
        //Поля, cwidth и cheight содержат количество клеток по
        //ширине и высоте игрового поля соответственно
        int cwidth, cheight;
        //Массив, для хранения данных о статусе
        //каждой клетки игрового поля
        array<bool, 2>^ fld;
        //Свойства:
        /*Индексированное свойство, получающее и задающее статус
каждой клетки на игровом поле. Оно поддерживает проверку допустимого
диапазона границ для индексов.*/
        public:  property bool Fld[int, int]{
                    bool get(int i, int j){
                        if (i>=0 && j>=0 &&
                            i<cWidth &&  j<cHeight)
                        {
                            return fld[i,j];
                        } else {
                            return false;
                        }
                    }
```

```
                void set(int i,
                        int j,
                        bool value)
                    {
                    if (i >= 0 && j>=0 &&
                        i<cWidth && j<cHeight)
                    {
                        fld[i,j] = value;
                    }
                }
            }
```

*/*Свойство, позволяющее задать и получить значение количества клеток по ширине игрового поля. Оно поддерживает проверку допустимых границ для задаваемых значений.*/*

```
        public: property int cWidth
            {
                int get()
                {
                    return cwidth;
                }
                void set(int value)
                {
                    //ширина игрового поля должна быть
                    //в пределах от 5 до 100 клеток
                    if (value < 5) {
                        cwidth = 5;
                    } else {
                        cwidth = value;
                        if (cwidth > 100) {
                            cwidth = 100;
                        }
                    }
                    NewArr(cwidth, cheight);
                }
            }
```

*/*Свойство, позволяющее задать и получить значение количества клеток по высоте игрового поля. Свойство поддерживает проверку допустимых границ для задаваемых значений.*/*

```
        public: property int cHeight
            {
                int get()
                {
                    return cheight;
                }
```

```
                        void set(int value)
                        {
                            //высота игрового поля должна быть
                            //в пределах от 5 до 100 клеток
                            if (value < 5)
                            {
                                cheight = 5;
                            }
                            else
                            {
                                cheight = value;
                                if (cheight > 100)
                                {
                                    cheight = 100;
                                }
                            }
                            NewArr(cwidth, cheight);
                        }
                    }
        //Методы
        //Выделение памяти под массив с заданными размерами
        private: void NewArr(int w, int h);
        //Метод, позволяющий задать одновременно количество
        //клеток по ширине и высоте игрового поля
        public: virtual void SetSize(int cW, int cH);
        //Метод, генерирующий следующее поколение клеток
        public: virtual void Next();
        //Метод, позволяющий узнать количество соседей
        //для заданной клетки
        private: bool GetNeighbour(int i,int j,int k);
        //Конструктор класса
        public: GameGeneration(int cw, int ch);
    };
}
```

```
///Файл «GameGeneration.cpp»
#include "GameGeneration.h"
using namespace GameLife;

GameGeneration::GameGeneration(int cw, int ch)
{
    SetSize(cw, ch); //Устанавливаем размеры игрового поля
}
```

```
//Выделяем память для массива, содержащего статусы клеток
void GameGeneration::NewArr(int w, int h){
    fld = gcnew array<bool, 2>(w, h);
    for (int i = 0; i < w; i++){
        for (int j = 0; j < h; j++){
            //По умолчанию "живых" клеток нет:
            fld[i,j] = false;
        }
    }
}
//Задание размеров игрового поля.
void GameGeneration::SetSize(int cW, int cH)
{
    //Обращение к свойствам гарантирует корректность заданных
    //размеров и выделение памяти под нужное количество клеток
    cWidth = cW;
    cHeight = cH;
}
//Получаем статус соседа для данной клетки. Число k (условное)
//определяет,какой именно сосед нас интересует
bool GameGeneration::GetNeighbour(int i,int j,int k)
{
    //Определяем индексы соседей для данной клетки. Если клетка
    //находится на самом верху, то в число соседей будут входить
    // клетки самого нижнего ряда и т.п.
    int pi = ((i-1)>=0)?i-1:(cWidth-1);
    int ni = ((i+1)<(cWidth-1))?(i+1):0;
    int pj = ((j-1)>=0)?(j-1):(cHeight-1);
    int nj = ((j+1)<cHeight-1)?(j+1):0;
    //Возвращаем статус нужного соседа
    switch (k)
    {
        case 0:{return Fld[i , pj];}//сосед слева
        case 1:{return Fld[ni, pj];}//сосед слева, снизу
        case 2:{return Fld[ni, j ];}//сосед снизу
        case 3:{return Fld[ni, nj];}//сосед справа, снизу
        case 4:{return Fld[i , nj];}//сосед справа
        case 5:{return Fld[pi, nj];}//сосед справа, сверху
        case 6:{return Fld[pi, j ];}//сосед сверху
        case 7:{return Fld[pi, pj];}//сосед слева, сверху
        default:{return false;}//задан неверный индекс соседа
    }
}
```

```
//Метод, генерирующий новое поколение клеток в соответствии
//с правилами игры. Для нового поколения используется временный массив
void GameGeneration::Next()
{
    array<bool, 2>^ newfld =
        gcnew array<bool, 2>(cWidth, cHeight);
    for (int i = 0; i < cWidth; ++i)
    {
        for (int j = 0; j < cHeight; ++j)
        {
            int c = 0;
            //Считаем количество "живых" соседей
            for (int k = 0; k<9; k++)
            {
                c += (GetNeighbour(i,j,k))?1:0;
            }
            //Заполняем новый массив в соответствии
            //с правилами игры
            if (Fld[i,j] && c>=2 && c<=3)
            {
                newfld[i,j] = true;
            }
            else
            {
                if (!Fld[i,j] && c==3)
                {
                    newfld[i,j] = true;
                }
                else
                {
                    newfld[i,j] = false;
                }
            }
        }
    }
    //Переписываем новые данные в основной массив
    for (int i = 0; i<cWidth; ++i)
    {
        for (int j = 0; j<cHeight; ++j)
        {
            Fld[i,j] = newfld[i,j];
        }
    }
}
```

Приведенный выше класс может содержать информацию об игре и генерировать новые поколения, но он не умеет изображать поле на экране. Для решения этой задачи создадим еще один класс, **GameField**, который будет являться наследником класса **GameGeneration**, способным выводить изображение на графическую поверхность.

```
//Файл «GameField.h»
#pragma once
#include "GameGeneration.h"

namespace GameLife
{
    using namespace System;
    using namespace System::Drawing;
    using namespace System::Drawing::Drawing2D;

    //Класс "GameField" ("Игровое поле") наследует от класса
    //"GameGeneration".  Данный класс позволяет
    //изображать клетки на некоторой
    //графической поверхности (Grpaphics).
    ref class GameField: public GameGeneration
    {
        //Некоторая графическая поверхность,
        //где будет отображаться игровое поле
    private: Graphics^ g;
        //конструктор
    public: GameField(int w, int h, Graphics^ g);
        //Метод, генерирующий изображение игрового поля
    public:  virtual void Paint();
        //Метод, позволяющий получить в структуре Point номер
        //клетки, которая содержит экранные координаты x и y
    private: Point^ GetBox(int x, int y);
        //Метод, помечающий клетку, содержащую координаты
        //cx и cy, и снимающий отметку с клетки,
        //содержащей координаты prevcx, prevcy
    public:  virtual void Mark(int prevcx, int prevcy,
                               int cx, int cy);
        //Метод устанавливает клетку, содержащую экранные
        //координаты x и y как "живую"
    public: virtual void Set(int x, int y);
        //Переопределенный метод, генерирующий новое поколение
        //клеток игрового поля
    public: virtual void Next() override;
```

```
        //Свойство для получения и задания графической поверхности
        //для отображения на нем игрового поля
    public: property Graphics^ Graph
    {
                Graphics^ get()
                {
                    return g;
                }
                void set(Graphics^ value)
                {
                    g = value;
                }
            }
    };
    //Структура, содержащая параметры игрового поля:
    //ширину, высоту всего поля в пикселях (w, h)
    //и отдельных клеток (wsz, hsz),
    // сдвиг от начала координат (x, y).
    //Значения структуры заполняются конструктором
    //по указанным параметрам:
    //графической поверхности и количеству клеток на игровом поле
    private value struct Box
    {
        public:
            int x;
            int y;
            int w;
            int h;
            int wsz;
            int hsz;
        public: Box(Graphics^ g, int cW, int cH)
            {
                RectangleF rect =
                            g->VisibleClipBounds;
                wsz = ((int)rect.Width) / cW;
                w = wsz * cW;
                hsz = ((int)rect.Height)/ cH;
                h = hsz*cH;
                x = (((int)rect.Width) - w )/2;
                y = (((int)rect.Height) - h )/2;
            }
    };
}
```

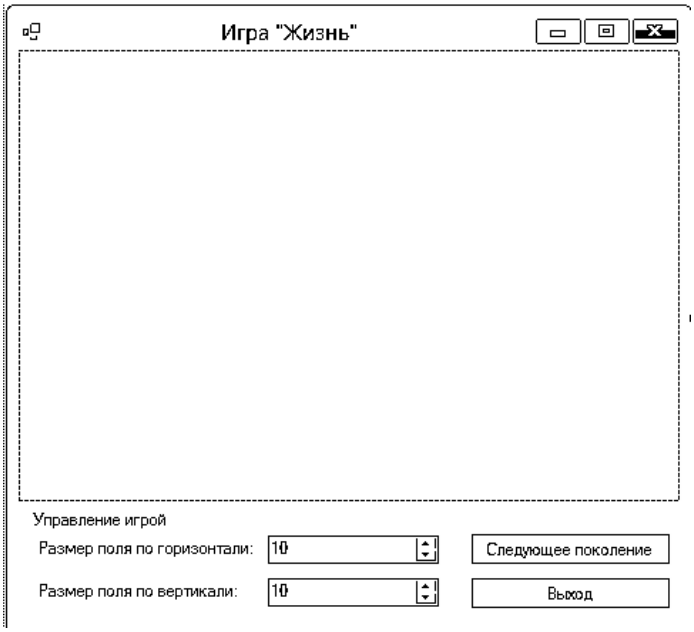
Реализация методов класса **GameField** приведена ниже:

```
//Файл «GameField.cpp»
#include "GameField.h"
using namespace GameLife;
//Конструктор класса GameField вызывает конструктор родительского
класса, а затем сохраняет в поле Graphics^ g графическую поверхность
GameField::GameField(int w, int h, Graphics^ g)
                        : GameGeneration(w, h)
{
    this->g = g;
}
//Метод прорисовки изображения
void GameField::Paint(){
    //Стираем все ранее существовавшие изображения
    g->Clear(Color::White);
    //Задаем ручку для прорисовки контуров
    Pen^ p = gcnew Pen(Color::Red);
    //Получаем координаты видимой области
    Box b(g, cWidth, cHeight);
    //Изображаем сетку на игровом поле.
    for (int i = b.x; i<b.w+b.x; i+=b.wsz){
        for (int j = b.y; j<b.h+b.y; j+=b.hsz){
            //Получаем координаты и размеры очередной клетки
            Rectangle r(i+1, j+1, b.wsz-1, b.hsz-1);
            //узнаем условные (по счету от левой верхней)
            //координаты клетки
            Point ^ p = GetBox(i,j);
            //Формируем кисть, цвет которой будет зависеть
            //от статуса клетки
            LinearGradientBrush ^lb;
            if (Fld[p->X,p->Y]){
                lb = gcnew LinearGradientBrush(r,
                            Color::OrangeRed,
                            Color::Coral, 90);
            } else {
                lb = gcnew LinearGradientBrush(r,
                            Color::LightGray,
                            Color::Gray, 90);
            }
            //формируем изображение клетки
            g->FillRectangle(lb, r);
        }
    }
}
```

```
//Получаем условные координаты (по счету) той клетки,
//которая содержит экранные координаты cx и cy
Point^ GameField::GetBox(int cx, int cy)
{
    Box b (g, cWidth, cHeight);
    Point^ p = gcnew Point(-1, -1);
    if (cx>=b.x && cy>=b.y &&
        cx<b.w+b.x && cy<b.h+b.y)
    {
        p->X = (cx-b.x) / b.wsz;
        p->Y = (cy-b.y) / b.hsz;
    }
    return p;
}
//Метод, помечающий клетку, содержащую координаты cx и xy,
//и снимающий отметку с клетки, содержащей координаты prevcx, prevcy
void GameField::Mark(int prevcx, int prevcy,
                     int cx, int cy)
{
    Point^ pnew = GetBox(cx, cy);
    Point^ pold = GetBox(prevcx, prevcy);
    if (pnew->Equals(pold))
    {
        //Изменения цвета не происходит, если и новые, и старые
        //координаты относятся к одной клетке
        return;
    }
    Box b(g, cWidth, cHeight);
    //Прямоугольные области соответствующие клеткам:
    //... которую обводим
    Rectangle nr(pnew->X*b.wsz+b.x+1,
                 pnew->Y*b.hsz+b.y+1,
                 b.wsz-1, b.hsz-1);
    //... у которой убираем обводку
    Rectangle or(pold->X*b.wsz+b.x+1,
                 pold->Y*b.hsz+b.y+1,
                 b.wsz-1, b.hsz-1);
    if (pold->X!=-1 && pold->Y!=-1)   {
        //Снимаем границу
        Pen^ open = gcnew Pen(Color::White);
        g->DrawRectangle(open, or);
        //Цвет заливки определяется статусом клетки
        //("живая" или нет)
        LinearGradientBrush^ lb;
```

```
        if (Fld[pold->X,pold->Y]){
            lb = gcnew LinearGradientBrush(or,
                                Color::OrangeRed,
                                Color::Coral, 90);
        }
        else
        {
            lb = gcnew LinearGradientBrush(or,
                                Color::LightGray,
                                Color::Gray, 90);
        }
        //Заливаем клетку вновь, чтобы не было белого ободка
        g->FillRectangle(lb, or);
    }
    //Для новой клетки изображаем красную границу
    if (pnew->X!=-1 && pnew->Y!=-1)
    {
        Pen^ npen = gcnew Pen(Color::Red);
        g->DrawRectangle(npen, nr);
    }
}
//Изменяем статус клетки ("живая" или нет).
//Параметры представляют собой
//экранные координаты, которые должна содержать нужная клетка.
void GameField::Set(int x, int y)
{
    Point ^p = GetBox(x, y);
    if (p->X !=-1 && p->Y != -1)
    {
        Fld[p->X, p->Y] = !Fld[p->X, p->Y];
    }
    this->Paint();
}
//Переопределенный метод генерации нового поколения, в задачу которого
//также входит и перерисовка игрового поля.
void GameField::Next()
{
    //Выполняем метод родительского класса
    GameGeneration::Next();
    //Перерисовываем изображение
    this->Paint();
}
```

Основное окно приложения, создаваемое средствами визуального программирования, может иметь примерно следующий вид.



Создать такое окно можно в режиме дизайнера с использованием стандартных элементов управления (панель, метки, кнопки, переключатели). Код окна формы приведен ниже.

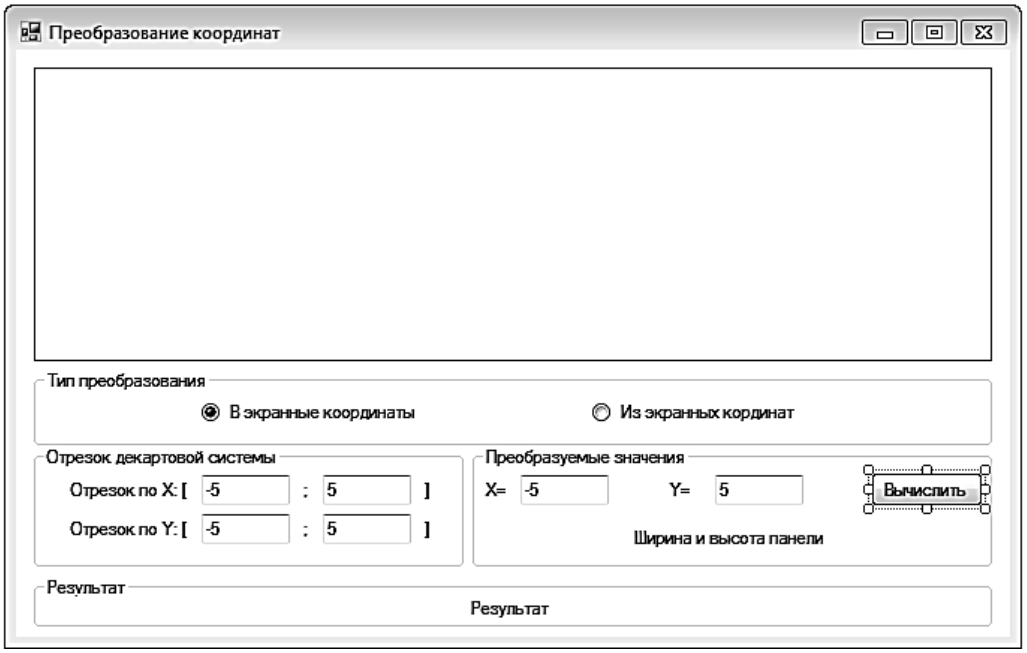
```
//Файл "Form1.h"
//Содержимое, сгенерированное в режиме дизайна,
//не отображается в целях экономии места.
#pragma once
#include "GameField.h"

namespace Form1Namespace
{
	//... часть автоматически созданного кода пропущено ...

	// Конструктор класса формы.
	Form1(void)
	{
		InitializeComponent();
		// Создаем объект «Игровое поле»
		gf = gcnew GameField(
			(int)numericUpDownW->Value,
			(int)numericUpDownH->Value,
			panel1->CreateGraphics());
		// Переменные, хранящие предыдущую позицию
		// курсора мыши при перемещении по панели
		lastX = lastY = -1;
	}
```

```
//... часть автоматически созданного кода пропущено ...
/// Декларация некоторых дополнительных полей
#pragma region MyDeclarations
    private: GameField^ gf;  //Объект «игровое поле»
    private: int lastX;
    private: int lastY;
#pragma endregion
//Обработчик события "Paint" для панели (panel1)
private: System::Void panel1_Paint(
    System::Object^  sender,
    System::Windows::Forms::PaintEventArgs^  e){
                    //Получаем графическую поверхность панели
                    //и передаем ее объекту «Игровое поле».
                    gf->Graph = panel1->CreateGraphics();
                    //Рисуем на поле изображение при помощи
                    //методов соответствующего класса
                    gf->Paint();
        }
//Обработчик события нажатия на кнопку «Выход»
private: System::Void button1_Click(System::Object^
sender, System::EventArgs^  e)    {
            Application::Exit();
        }
//Обработчик изменения размера панели (SizeChanged)
private: void panel1_SizeChanged(
    System::Object^  sender,
    System::EventArgs^ e)
    {
        //Производим обновление панели (вызов обработчика Paint)
        panel1->Refresh();
    }
//Обработчик события «движение мыши» (MouseMove)
private: void panel1_MouseMove(
        System::Object^ sender,
        System::Windows::Forms::MouseEventArgs^  e)
{
    //Вызываем метод объекта «Игровое поле» для пометки
    //клетки, над которой располагается мышь
    gf->Mark(lastX, lastY, e->X, e->Y);
    //Запоминаем текущие координаты мыши для снятия
    //пометки с клетки, когда курсор с нее уйдет
    lastX = e->X;
    lastY = e->Y;
}
```

```
//Обработчик события, наступающего при изменении значений,
//задающих размеры поля
private: System::Void numericUpDown_ValueChanged(
             System::Object^ sender,
             System::EventArgs^ e)
{
      //Устанавливаем заданные размеры в параметрах
      //объекта «Игровое поле»
       gf->SetSize((int)numericUpDownW->Value,
                   (int)numericUpDownH->Value);
       //Обновляем панель, вызывая метод ее перерисовки — Paint
       panel1->Refresh();
}
//Обработчик события «Щелчок мышью по панели»
private: void panel1_MouseClick(
         System::Object^ sender,
         System::Windows::Forms::MouseEventArgs^ e)
{
       gf->Set(e->X, e->Y);
}
//Обработчик нажатия на кнопку «Следующее поколение»
private: void button2_Click(System::Object^ sender,
                            System::EventArgs^  e)
         {
             //Производим генерацию и отображения нового
             //вида для игрового поля
             gf->Next();
         }
     };
}
```

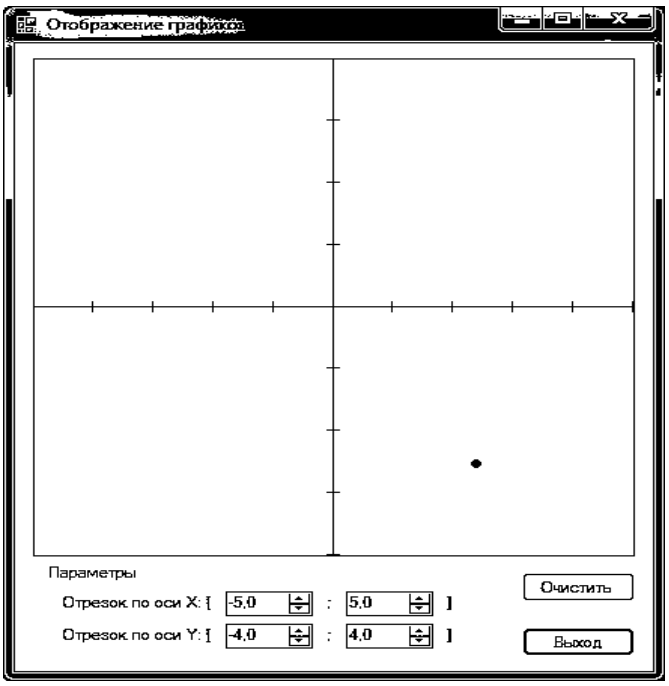
В. Задания для самостоятельного выполнения

1. Требуется написать программу с графическим интерфейсом пользователя, которая будет осуществлять преобразование координат из экранных в декартовые и обратно. Примерный внешний вид программы представлен на рисунке ниже:



Методы преобразования должны быть реализованы в отдельном классе.

2. Требуется создать приложение с графическим интерфейсом пользователя, которое бы позволяло выводить в контейнер (панель) изображение декартовой системы координат. Примерный вид приложения приведен ниже:



Требования к программе.

А. Программа должна содержать отдельный управляемый ссылочный класс для преобразования координат точек из экранных в декартовые и обратно (см. задание 1).

Б. Программа должна содержать отдельный управляемый ссылочный класс для отображения декартовой системы координат. Этот класс должен использовать методы класса-преобразователя.

В. В декартовой системе координат отображаются деления в соответствии с установленными в параметрах программы границами отрезков.

Г. Программа осуществляет контроль ввода границ, не позволяя установить левую границу больше правой и наоборот.

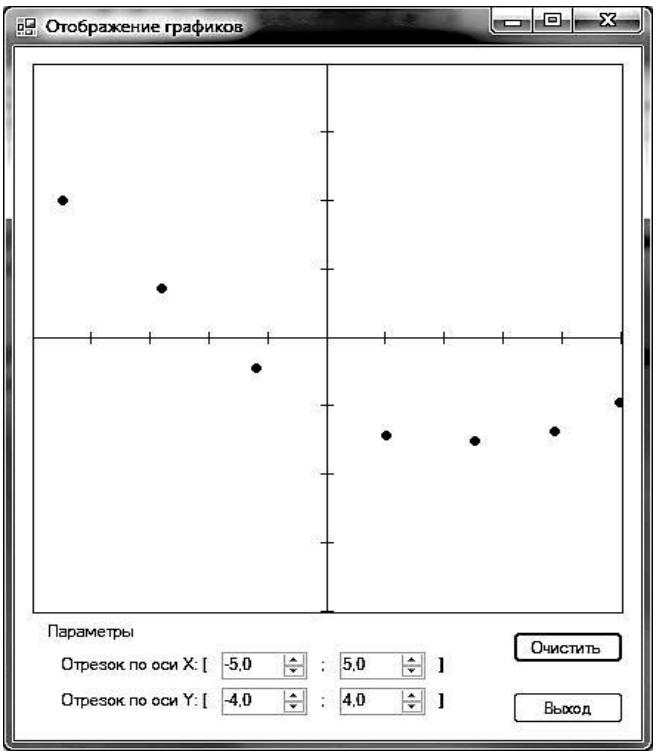
Д. При изменении размеров окна размер панели изменяется пропорционально, а содержимое панели перерисовывается автоматически так, чтобы занимать все предоставленное пространство.

3. Добавить в проект задания 2:

   А. Управляемый ссылочный класс для рисования точки на координатной плоскости. Класс должен содержать поля для хранения координат точки (x и y в декартовой системе координат), свойства для доступа к ним. Класс должен содержать методы для прорисовки своей точки (в виде хорошо заметного кружка) на заданной поверхности рисования **`Graphics`** (по аналогии с прорисовкой осей декартовой системы координат).

   Б. Добавить обработчики событий для панели. При щелчке мышью на ней необходимо прорисовывать точку.

4. Требуется расширить функционал программы примера 3, задав возможность вывода на панель нескольких точек при щелчках мыши в разных местах. Также необходимо задать функциональность для кнопки «Очистить», которая должна стирать с панели все точки.



Требования к программе:

А. Необходимо создать класс "Набор точек" (**`DecartPoints`**) для работы с несколькими точками. Класс должен быть наследником класса, который выводит на экран декартовую систему координат (задание 2).

Б. В классе **`DecartPoints`** необходимо создать коллекцию (**`List`**) из пространства имен **`System::Collections::Generic`** для хранения объектов типа **`DecartPoint`** (класс из задания 3, рисующий одну точку).

В. При описании класса **`DecartPoints`** переопределить методы вывода изображения на экран и обновления объекта.

Г. Позиции установленных точек должны сохраняться в заданных декартовых (не экранных) координатах при изменении размеров панели.

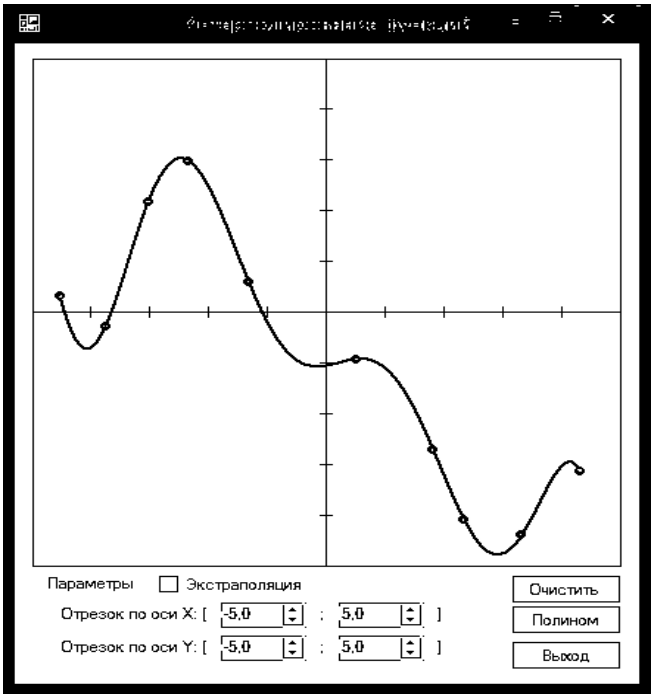
Д. В классе **`DecartPoints`** добавить методы проверки существования некоторой точки в коллекции.

Е. В методе добавления точки в коллекцию проверить наличие добавляемой точки. В случае если точка с такой же (с некоторой точностью) абсциссой в коллекции уже существует, новую точку в коллекцию не добавлять.

Ж. Добавить в класс **`DecartPoints`** метод удаления некоторой точки (по двум ее координатам) из коллекции.

3. Добавить в класс **DecartPoint** (задание 3) метод **CompareTo** для реализации интерфейса **IComparable**, чтобы имелась возможность сравнения объектов **DecartPoint**. Реализовать сравнение точек по их x-координате. Две точки следует считать одинаковыми, если их абсциссы совпадают с определенной точностью (например, 0,01 — задать полем класса). Реализация этого интерфейса позволит создавать сортированные списки объектов **DecartPoint**, например, такие как **SortedList<DecartPoint^>**.

5. Разработать классы **Polynom** и **NewtonPolynom** (либо **LagrangePolynom**) для работы с произвольными полиномами и интерполяционными полиномами Ньютона (либо Лагранжа) соответственно.



А. Реализовать класс **DecartPolynom**, как наследник класса **DecartPoints**, для рисования полинома Ньютона на панели при помещении на нее точек. Переопределить в этом классе метод **Paint**.

Б. Добавить возможность удаления точек с панели при правом клике мыши в окрестности этой точки. При любом изменении количества точек на панели полином должен автоматически перестраиваться.

В. Добавить в окно кнопку «Полином» для получения текстового представления полинома в окне информации.

III. Создание приложений с несколькими потоками. Синхронизация потоков в C++/CLI.

А. Теоретическая справка

1. Создание и использование потоков.

Поток — это независимый путь исполнения программного кода, способный выполняться одновременно с другими потоками. При запуске программы, написанной на языке C++, операционной системой формируется процесс, в котором автоматически создается единственный («главный») поток. Все действия выполняются одним потоком строго последовательно. Для выполнения нескольких действий параллельно (например, на разных процессорах, или, хотя бы, на одном процессоре в режиме разделения времени), необходимо добавить в процесс другие потоки.

Для создания в программе дополнительных потоков можно использовать класс Thread. Он создает и контролирует поток, задает приоритет и возвращает его статус.

Таблица III-1. Класс Thread

Конструкторы	Описания
Thread(ThreadStart^ start)	Инициализирует новый экземпляр класса Thread. Делегат ThreadStart^ start указывает на метод, который вызывается при запуске потока.
Thread(ParameterizedThreadStart^ start)	Инициализирует новый экземпляр класса Thread. При этом указывается делегат ParameterizedthreadStart^ start, позволяющий объекту быть переданным в поток при его запуске.
Свойства	**Описания**
property bool IsAlive	Возвращает значение, показывающее статус выполнения текущего потока. Свойство содержит значение true, если этот поток был запущен и не был завершен; в противном случае свойство содержит значение false.
property bool IsBackground	Получает или задает значение, показывающее, является ли поток фоновым.
property ThreadPriority Priority	Получает или задает значение, указывающее на планируемый приоритет потока. Значение свойства может быть одним из следу-

Свойства	Описания
	ющих: *Lowest:* Низший приоритет. Выполнение потока производится в последнюю очередь. *BelowNormal:* Приоритет ниже нормального. Выполнение производится после потоков с приоритетом Normal и до потоков с приоритетом Lowest. *Normal:* Нормальный приоритет. Выполнение производится после потоков с приоритетом AboveNormal и до потоков с приоритетом BelowNormal. Это значение установлено по умолчанию. *AboveNormal:* Приоритет потока выше нормального. Выполнение производится после потоков с приоритетом Highest и до потоков с приоритетом Normal. *Highest:* Наивысший приоритет. Выполнение потока производится в первую очередь.
property ThreadState ThreadState	Возвращает значение, содержащее состояния текущего потока. Оно может принимать одно значений: *Running:* Поток был запущен, не заблокирован, и нет ожидающего исключения ThreadAbortException. *StopRequested:* Поток получает запрос на остановку. Предназначено только для внутреннего использования. *SuspendRequested:* Запрашивается приостановка работы потока. *Background:* Поток выполняется как фоновый, в проти-

Свойства	Описания
	воположность потокам переднего плана. Это состояние управляется заданием свойства Thread::IsBackground. *Unstarted:* Метод Thread::Start не был вызван для потока. *Stopped:* Поток был остановлен. *WaitSleepJoin:* Поток заблокирован. Это может произойти в результате вызова методов Thread::Sleep или Thread::Join, а также в результате запроса блокировки, например при вызове методов Monitor::Enter или Monitor::Wait, или в результате ожидания объекта синхронизации потока, такого как ManualResetEvent. *Suspended:* Поток был приостановлен. *AbortRequested:* Метод Thread::Abort был вызван для потока, но поток еще не получил исключение System.Threading::ThreadAbortException, которое попытается завершить его. *Aborted:* Состояние потока включает в себя значение AbortRequested, и поток теперь не выполняет работу, но его состояние еще не изменилось на Stopped.

Методы	Описания
void Abort()	Вызывает исключение ThreadAbortException в вызвавшем его потоке для того, чтобы начать процесс завершения потока. Вызов данного метода обычно завершает поток.
void Join()	Блокирует вызывающий поток до завершения этого потока, продолжая отправлять стандартные сообщения.
static void Sleep(int millisecondsTimeout)	Блокирует текущий поток на заданное количество миллисекунд.

Методы	Описания
void Start()	Запускает поток на выполнение
void Start(Object^ parameter)	Запускает поток на выполнение, а также передает объект с данными, используемыми методом в потоке.

2. Синхронизация потоков.

Синхронизация потоков — основная трудность, с которой сталкиваются программисты, начинающие создавать многопоточные приложения.

Если в программе присутствует только один поток, он использует все ресурсы, к которым программа имеет доступ, в той последовательности, которая соответствует логике ее работы. Но потоки, работающие параллельно, могут пытаться обращаться одновременно к одним и тем же ресурсам, что может привести к неправильной работе приложений. Чтобы избежать ошибок, необходимо контролировать доступ к общим для нескольких потоков ресурсам. Обычно в таких случаях используются те или иные механизмы синхронизации.

Для синхронизации работы потоков можно использовать специальные классы, входящие в библиотеку .NET:

Таблица III-2

Класс	Описание
AutoResetEvent	Блокирует поток до установки события другим потоком.
ManualResetEvent	Блокирует один или несколько потоков до установления события другим потоком.
Interlocked	Организует совместную работу потоков при выполнении простых операций с числами: инкремент, декремент, обмен, обмен и сравнение.
Monitor	Гарантирует последовательный доступ нескольких потоков к заблокированному объекту.
Mutex	Гарантирует последовательный доступ нескольких потоков к заблокированному объекту и, в отличие от Monitor, может быть использован для синхронизации работы разных приложений.
ReaderWriterLock	Обеспечивает параллельное чтение ресурса несколькими потоками, но строго последовательную запись информации в ресурс потоками поочередно.

Наиболее интересным для нас из них является «Monitor». Он позволяет осуществлять синхронизацию потоков на уровне доступа к объектам. Все методы класса являются статическими и не требуют создания его экземпляров.

Таблица III-3. Класс Monitor

Методы	Описания
static void Enter(Object^ obj)	Получает эксклюзивную блокировку указанного объекта. Параметр obj — объект ссылочного типа, для которого получается блокировка монитора.
static bool TryEnter (Object^ obj, int millisecondsTimeout)	Пытается получить эксклюзивную блокировку указанного объекта. Параметр obj — объект ссылочного типа, для которого получается блокировка монитора. Если по истечении указанного в миллисекундах времени блокировка не была получена, метод возвращает false и передает управление в поток, позволяя ему выполнить другие операции. В противном случае метод возвращает значение true.
static void Exit(Object^ obj)	Освобождает эксклюзивную блокировку указанного объекта. Параметр obj — объект ссылочного типа, для которого снимается блокировка монитора. Должен быть ранее заблокирован методом Enter.
static void Pulse(Object^ obj)	Уведомляет поток в очереди готовности об изменении состояния объекта с блокировкой. Параметр obj — это объект ссылочного типа, которого ожидает поток.
static void PulseAll (Object^ obj)	Уведомляет все ожидающие потоки об изменении состояния объекта. Параметр obj — это объект ссылочного типа, которого ожидают потоки.
static bool Wait (Object^ obj)	Освобождает блокировку объекта obj и блокирует текущий поток до тех пор, пока тот не получит блокировку снова. Возвращаемое значение — true, если вызывающий поток заново получил блокировку заданного объекта. Этот метод не осуществляет возврат, если блокировка вновь не получена.

Методы	Описания
static bool Wait(Object^ obj, int millisecondsTimeout)	Освобождает блокировку объекта obj и блокирует текущий поток до тех пор, пока тот не получит блокировку снова. Если указанные временные интервалы (millisecondsTimeout) истекают, поток встает в очередь готовности.

3. Делегаты. Модель событий.

В некоторых задачах требуется отображать ход выполнения длительного вычислительного процесса или выводить результат, получаемый в отдельном потоке, на экран после завершения всех расчетов. Однако привычная модель вывода данных в этом случае работать уже не будет, поскольку действия основного потока (вывод результата) могут выполниться раньше, чем будет посчитан сам результат в другом потоке.

Для решения этой проблемы можно воспользоваться событийной моделью. Событие представляет собой сообщение, посылаемое объектом, чтобы уведомить о совершении какого-либо действия. Объект, вызывающий событие, называется отправителем события. Объект, захватывающий событие и реагирующий на него, — получателем события. При обмене событиями классу отправителя событий не известен объект или метод, который будет получать (обрабатывать) сформированные отправителем события, поэтому требуется наличие некоторого механизма, позволяющего отправителю выполнять действия, реализованные в классе-получателе (так называемый обратный вызов). Для этого в .NET определен специальный тип «Delegate», обеспечивающий функциональные возможности указателя функции.

Пример описания делегата на языке C++/CLI:

```
delegate void SomeHandler(Object^ someParametr);
```

Делегат является классом, который может хранить ссылку на метод. Как видно из примера, класс делегата, в отличие от прочих, имеет сигнатуру и может хранить ссылки только на методы, соответствующие этой сигнатуре. Таким образом, делегат эквивалентен типобезопасному указателю на функцию, который можно испольховать для обратного вызова. Объявление делегата является достаточным для определения класса делегата. Объявление предоставляет сигнатуру делегата, а среда CLR обеспечивает его реализацию.

Чтобы класс мог **ПОРОДИТЬ СОБЫТИЕ**, необходимо подготовить:

- делегат события;
- объявление события;
- определение метода, порождающего событие.

Член-событие определяется с помощью ключевого слова event:

```
public: event SomeHandler^ OnSomeEvent;
```

После такого определения необходимо установить, когда следует инициировать событие. Для этого его в необходимый момент нужно породить:

```
Object^ someObject;
//...
OnSomeEvent(someObject);
```

Для **ПРИЕМА СОБЫТИЯ** в приложении необходимо:

- предоставить обработчик событий (метод обработки), который выполняет логику программы в ответ на событие,
- зарегистрировать обработчик для источника событий. Этот процесс также называют привязкой событий.

```
//...
//Объект класса, порождающего события
EventRaiser ^m = gcnew EventRaiser ();
//Привязываем событие OnSomeEvent к обработчику
//класса ThisClass — методу Met
m->OnSomeEvent += gcnew SomeHandler(
                  this,
                  &ThisClass::Met
                                  );
//...
//Сам обработчик
public void Met(Object^ someParametr)
{
     //...
}
```

Б. Примеры

ПРИМЕР №1. Следующая программа будет выводить на экран символ «a» основным потоком, и символ «b» — дополнительно созданным потоком. Таким образом, на экране должны будут чередоваться указанные буквы.

```
using namespace System;
//Подключение пространства имен для работы с потоками
using namespace System::Threading;
ref class MyClass{
public:
    void WriteA();
    void WriteB();
};
//Метод выводит 100 букв "a"
void MyClass::WriteA()
{
    for (int i = 0; i<100; ++i){
        Console::Write("a");
    }
}
//Метод выводит 100 букв "b"
void MyClass::WriteB()
{
    for (int i = 0; i<100; ++i){
        Console::Write("b");
    }
}
int main()
{
    //Создаем объект нашего класса
    MyClass mc;
    //Формируем объект ThreadStart, для которого указываем,
    //какой объект (mc) и какой метод (MyClass::WriteB)
    //будет выполнять поток.
    //Знак % означает получение адреса ссылочного объекта.
    //Знак & - получение адреса объекта, не являющегося ссылочным,
    //в данном случае – метода WriteB класса MyClass.
    ThreadStart^ ts = gcnew ThreadStart(%mc,

    &MyClass::WriteB);
    //Формируем объект "Поток", которому передаем
    //параметры запуска
    Thread ^t = gcnew Thread(ts);
    //Запускаем новый поток, который будет выводить символ "b"
    t->Start();
    //Запускаем метод, который выведет символ "a" в главном потоке
    mc.WriteA();
}
```

В результате работы программы на экране появится примерно следующее:

aaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaa aaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaa aaaaa

В данном примере невозможно передать параметры в метод, запускаемый в отдельном потоке. Это можно сделать, если вместо класса **ThreadStart** использовать **ParameterizedThreadStart**. Тогда в нашем классе можно убрать один из методов, а в оставшийся передать параметр — строку, которую требуется вывести.

```
using namespace System;
//Подключение пространства имен для работы с потоками
using namespace System::Threading;
ref class MyClass
{
public:
    void Write(Object^ obj);
};
///<summary>Метод выводит 100 символов</summary>
///<param name="obj">obj - объект, через который можно
///передать любое значение, а в нашем случае будет
///передаваться строка.</param>
void Write(Object^ obj)
{
    //Преобразуем объект obj к типу String.
    //Отметим, что данная операция является небезопасной
    //и вызовет ошибку времени выполнения, если фактически
    //будет передан объект какого-либо иного типа.
    //Это следует учитывать при создании реальных проектов.
    String^ s = (String^)obj;
    //В данном случае как вариант можно
    //применить конструкцию: String^ s = obj->ToString();
    for (int i = 0; i<100; ++i){
        Console::Write(s);
    }
}
int main()
{
    //Создаем объект нашего класса
    MyClass mc;
```

```
    //Формируем объект ParametrizedThreadStart,
    //для которого указываем,
    //кто (объект mc) и что (метод Write класса MyClass)
    //будет выполнять.
    ParameterizedThreadStart^ ts = gcnew
        ParameterizedThreadStart(%mc,
                                 &MyClass::Write);
    //Формируем объект "Поток", которому передаем
    //параметры запуска
    Thread ^t = gcnew Thread(ts);
    //Запускаем новый поток, который будет выводить символ "b"
    t->Start("b");
    //Запускаем метод, который выведет символ "a" в главном потоке
    mc.Write("a");
}
```

Пример №2. В некоторых случаях при параллельной работе нескольких потоков могут возникать неприятные побочные эффекты. Рассмотрим следующую небольшую программу, которая имитирует взаимодействие клиента с банковским счетом. Пусть имеется класс **Bank**, в конструкторе которого указывается стартовая сумма на счете клиента. Имеется метод **Transact**, позволяющий изменить сумму, находящуюся на счете и метод **PrintSum**, печатающий текущий баланс.

```
#include <cmath>
#pragma once
using namespace System;
using namespace System::Threading;
ref class Bank {
private:
    float sum;
public:
    //Конструктор задает начальное значение для суммы на счете
    Bank(float start_sum)
    {
        sum = start_sum;
    }
    //Этот метод позволяет узнать размер текущего остатка на счете
    float Bank::GetSum()
    {
        return sum;
    }
```

```
    //Метод устанавливает новое значение для остатка на счете
    void Bank::SetSum(float s)
    {
        sum = s;
    }
    //Метод проверяет, достаточно ли денег для списания, и либо
    //разрешает снять деньги со счета, либо выводит
    //сообщение об ошибке.
    //Также метод позволяет пополнить счет.
    void Bank::Transact(Object ^sum_delta)
    {
        //Преобразуем параметр к вещественному типу
        float delta =
            Single::Parse(sum_delta->ToString());
        //Узнаем текущий остаток
        float s = GetSum();
        if ((delta<0 && s>abs(delta)) || delta>=0)
        {
            //Если клиент снимает деньги (delta <0),
            //проверяем, что на счете
            //достаточно средств и устанавливаем новое значение.
            // При delta > 0 деньги вносятся на счет.
            SetSum(s + delta);
        }
        else
        {
            //Если средств на счете недостаточно
            Console::WriteLine("Запрошенная сумма ("
                +abs(delta)+"р.) больше имеющейся ("
                +GetSum()+"р.)");
        }
    }
    //Выводим информацию о текущем остатке
    void Bank::PrintSum()
    {
        Console::WriteLine("Остаток составляет: "
                                        +sum+"р.");
    }
};
```

Еще раз заметим, что метод **Transact** проверяет, не является ли значение, переданное через параметр (объектный тип, приведенный к вещественному числу), больше суммы, которая лежит на счете и не дает возможности клиенту взять в кредит.

Воспользуемся этим классом из нескольких потоков.

```
#include "Bank.h"
using namespace System;
using namespace System::Threading;

int main()
{
    Bank b(1567.87);
    ParameterizedThreadStart^ ps =
            gcnew ParameterizedThreadStart(
                                %b,
                                &Bank::Transact);
    Thread^ t = gcnew Thread(ps);
    //Выполняем транзакцию в дополнительном потоке
    //и добавляем на счет 1000р.
    t->Start(1000.);
    //Пытаемся снять 2000 р.
    b.Transact(-2000);
    b.PrintSum();//Печатаем полученный результат.
    Thread::Sleep(5000);//Пауза, чтобы увидеть результат
}
```

В некоторых случаях, после выполнения данной программы на компьютере, мы получим вполне ожидаемые результаты:

```
Остаток составляет 567,87р.
```

или

```
Запрошенная сумма (2000р.) больше имеющейся (1567,87р.)
Остаток составляет: 2567,87р.
```

в зависимости от того, какой из потоков выполнится раньше.

Однако это не единственный возможный случай, и компьютер может «порадовать» неожиданными заявлениями:

```
Запрошенная сумма (2000р.) больше имеющейся (2567,87р.)
Остаток составляет: 2567,87р.
```

Что же стало причиной непредсказуемой работы программы? С каких пор 2000 > 2567,87? Почему клиенту не дали возможность снять

имеющиеся у него деньги? Все дело в том, что в данной программе мы не можем предсказать очередность выполнения действий в разных потоках.

В последнем случае, очевидно, произошло следующее.

- Поток №1 пытается пополнить счет на 1000р. Он узнает текущий остаток при помощи метода **GetSum** и записывает его в локальную переменную s (1567,87р).
- Поток №2 одновременно пытается снять со счета 2000р. Он также узнает текущий остаток при помощи метода **GetSum** и записывает его в свою локальную переменную **s** (1567,87р.).
- Поток №1 проверяет условия и, поскольку **delta** положительна, осуществляет сложение, а результат записывает на счет методом **SetSum** (2567.87р.).
- Поток №2 проверяет условие, в котором участвует его локальная переменная **s**, значение которой не менялось и составляет 1567,87р. Это значение меньше запрошенной суммы и поэтому метод генерирует отказ. В него он вписывает абсолютное значение **delta** (2000) и остаток на счете, который получает с помощью метода **GetSum**, однако, поскольку остаток уже изменен первым потоком, результат выполнения получается 2567,87р.!

Чтобы исправить ситуацию, необходимо синхронизировать работу потоков. Необходимо сделать так, чтобы после начала транзакции одним потоком, другой не имел возможности работать со счетом, пока первый не закончит свою работу. Это можно реализовать при помощи класса **Monitor** и его статических методов **Enter** и **Exit**, открывающих и завершающих синхронизированную секцию. В качестве параметра этим методам передается объект ссылочного типа, который разделяется несколькими потоками. В нашем случае, таким объектом могла бы быть переменная **sum**, но, к сожалению, она является типом-значением. Поэтому нам придется воспользоваться дополнительной, специально введенной переменной типа **Object^**.

Класс **Bank** примет следующий вид:

```
#include <cmath>
#pragma once
using namespace System;
using namespace System::Threading;
```

```
ref class Bank
{
private:
    float sum;
    static Object^ comm = gcnew Object();
public:
    ///...
    void Bank::Transact(Object ^sum_delta){
        float delta = Single::Parse(
                        sum_delta->ToString());
        Monitor::Enter(comm);
        try {
            float s = GetSum();
            if ((delta<0 && s>abs(delta))
                            || delta>=0){
                SetSum(s + delta);
            } else {
                Console::WriteLine("Запрошенная "
                +"сумма ("+abs(delta)
                +"р.) больше имеющейся ("
                +GetSum()+"р.)");
            }
        }
        finally {
            Monitor::Exit(comm);
        }
    }
    ///...
};
```

Теперь первый поток, зайдя в критическую секцию, заблокирует объект comm для других потоков, до тех пор, пока не будет вызван метод **Exit** класса **Monitor** для этого же объекта. Второй поток, дойдя до критической секции, будет ждать освобождения объекта comm. Таким образом, в данной секции потоки будут работать строго последовательно. Здесь следует учесть, что размер синхронизированных блоков должен быть как можно меньше, поскольку в них исчезает параллелизм, а значит, снижается производительность программы. Обратите внимание, что метод **Exit** должен находиться в блоке **finally** для гарантии того, что он будет выполнен даже в случае генерации неожиданных исключений. В противном случае есть риск зависания всех потоков, ожидающих снятия блокировки.

Пример №3. Пусть у нас имеется следующий класс:

```
//Файл "EventRaiser.h"
#pragma once
using namespace System;
using namespace System::Threading;
ref class EventRaiser {
private:
    int result;
public:
    void Calc(){//Метод, выполняющий вычисления
        result = 0;
        for (int i = 1; i<=100; i++){
            result += i;
        }
    }
    property int Result {//Свойство, возвращающее результат
        int get(){
            return result;
        }
    }
};
```

Воспользоваться им из основного потока программы легко:

```
//Файл "main.cpp"
#include "EventRaiser.h"
int main(){
    EventRaiser er;//Создаем экземпляр
    er.Calc();//Выполняем вычисления
    Console::WriteLine(er.Result);//Выводим результат
    Thread::Sleep(5000);//Пауза 5 сек., чтобы увидеть результат.
    return 0;
}
```

После выполнения программы на экране будет ожидаемый результат: «5050». Попытаемся выполнить вычисления в отдельном потоке.

```
//Файл "main.cpp"
#include "EventRaiser.h"
int main(){
    EventRaiser er;
    //Запускаем вычисления в дополнительном потоке
    ThreadStart^ ts = gcnew ThreadStart(%er,
                                        &EventRaiser::Calc);
    (gcnew Thread(ts))->Start();
```

```
    Console::WriteLine(er.Result); //Выводим результат
    Thread::Sleep(5000); //Пауза для просмотра результата
    return 0;
}
```

При выполнении этой программы на экране в некоторых случаях могут возникать иные значения, и даже «0». Действительно, после запуска второго потока методом **Start**, вычисления начинают выполняться параллельно с кодом, записанным в функции **main**. Таким образом, строчка с выводом результата может быть вызвана в процессе вычислений или до их начала. В приведенном примере лучше уведомить класс, который получает результат, об окончании вычислительного процесса при помощи механизма событий.

Внесем в программу некоторые изменения.

Класс, вызывающий события:

```
//Файл "EventRaiser.h"
#pragma once
using namespace System;
using namespace System::Threading;

//Объявляем делегат, определяющий сигнатуру метода
delegate void ResultHandler(int result);

ref class EventRaiser
{
private:
    int result;
public:
    //Определяем событие "По завершению",
    event ResultHandler^ OnFinish;
    //Выполнить вычисления
    void Calc(){
        result = 0;
        for (int i = 1; i<=100; i++)
        {
            result += i;
        }
        //Генерируем событие для подписанных слушателей.
        //При этом у них  сработает
        //метод, объявленный в качестве обработчика.
        OnFinish(result);
    }
```

```
    //Свойство, возвращающее результат
    property int Result{
        int get(){
            return result;
        }
    }
};
```

Класс, получающий и обрабатывающий события:

```
//Файл "EventCatcher.h"
#include "EventRaiser.h"
#pragma once
ref class EventCatcher {
private: EventRaiser ^er;
public:
    //Функция имеет сигнатуру, соответствующую описанному в классе
    //EventRaiser делегату
    void Show(int res){
        //Выводим на экран значение параметра
        Console::WriteLine(res);
        //Пауза для просмотра результата до закрытия окна.
        Thread::Sleep(5000);
    }
    EventCatcher(void)
    {
        //Создаем экземпляр класса EventRaiser
        er = gcnew EventRaiser();
        //И подписываемся на событие OnFinish,
        // указывая при этом обработчик
        er->OnFinish += gcnew ResultHandler(this,
                        &EventCatcher::Show);
    }
    void Start()
    {
        //Запускаем процесс вычислений в отдельном потоке
        //(как и ранее)
        ThreadStart^ ts = gcnew ThreadStart(
                        er,
                        &EventRaiser::Calc
                                        );
        Thread^ t = gcnew Thread(ts);
        t->Start();
    }
};
```

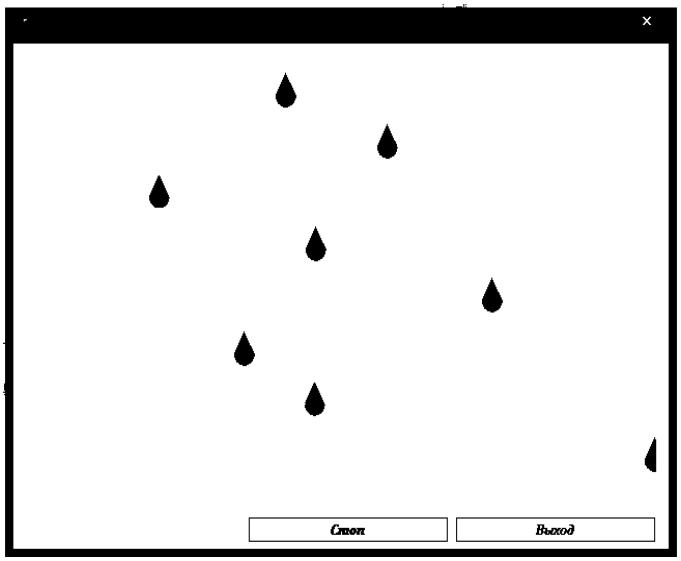
```
//Файл "main.cpp" — точка входа в программу
#include "EventCatcher.h"
int main()
{
    EventCatcher er;
    er.Start();
    return 0;
}
```

В этом примере после окончания цикла в методе **Calc** класса **EventRaiser** генерируется событие, которое вызывает исполнение обработчика **Show** класса **EventCatcher**. Теперь мы можем быть уверены в том, что вывод результата осуществится строго после выполнения вычислений.

Пример №4. Если вы создаете программу, содержащую графический интерфейс, то обращение к графическим элементам управления (например, присвоение текста метке), можно осуществлять только из того потока, в котором этот элемент был создан. В случае, когда обработчик события, в котором идет обращение к графическим компонентам, выполняется каким-то другим потоком, возможно возникновение исключительных ситуаций. Чтобы избежать этого, необходимо пользоваться следующим механизмом:

```
delegate void ProgressHandler(int proc);
//...
private: void SomeMethod(int proc){
    //Метод пытается изменить значение (свойство «Value»)
    //у графического элемента ProgressBar
    if (!progressBar1->InvokeRequired){
        //Если метод вызван из того же потока, в котором создан
        //progressBar1, то устанавливаем его значение
        progressBar1->Value = proc;
    } else {
        //в противном случае обращаемся к методу Invoke для вызова
        //метода из своего (для ProgressBar) потока. С этой целью
        //пользуемся делегатом, объявленным выше для этого метода
        Invoke(gcnew ProgressHandler(this,
                    &Form1::MyMatrixProcess), proc);
    }
}
```

Пример №5. Рассмотрим еще одну программу, работающую с потоками, на примере приложения, выводящего на экран изображение падающих дождевых капель (см. рисунок ниже).



Изображение каждой падающей капли, а также весь «дождь» в целом будет прорисовываться отдельным потоком. Если бы потоки не были использованы, приложение бы «зависло», и с его окном невозможно было бы полноценно работать.

```
//Файл "Form1.h"
#pragma once
#include "FallRain.h"
namespace Form1Namespace
{
    using namespace Rain;

    ///...часть автоматически сгенерированного кода пропущена

    public ref class Form1 :
        public System::Windows::Forms::Form
    {
    public:
        //Конструктор
        Form1(void)
        {
            InitializeComponent();
            //Создаем объект рабочего класса
            fr = gcnew FallRain(
                        panel1->CreateGraphics() );
        }
    protected:
```

```
        // Деструктор
        ~Form1() {
            if (fr) {//Если объект класса "FallRain" был создан
                delete fr;//Удаляем его, останавливая потоки
            }
            if (components){
                delete components;
            }
        }
    //... часть автоматически сгенерированного кода пропущена
    //Указатель на объект рабочего класса "Дождь"
    FallRain^ fr;
    //Обработчик нажатия на кнопку «Выход»
    private: System::Void exitBtn_Click(
            System::Object^  sender,
            System::EventArgs^  e) {
                //При выходе из программы удаляем объект
                //рабочего класса «Дождь», останавливая,
                // тем самым, все созданные фоновые потоки.
                delete fr;
                Application::Exit();
            }
    //Обработчик нажатия кнопки «Старт/Стоп»
    private: System::Void startBtn_Click(
            System::Object^  sender,
            System::EventArgs^  e) {
            //Если кнопка называется "Старт"
            if (startBtn->Text->Equals("Старт")){
                //переименовываем ее
                startBtn->Text = "Стоп";
                //и запускаем "дождь" с помощью метода
                //Start() класса "FallRain"
                fr->Start();
            } else {
                //в противном случае,
                //переименовываем кнопку обратно и...
                startBtn->Text = "Старт";
                //... останавливаем "дождь"
                fr->Stop();
            }
        }
    };
}
```

```
///Файл "Drop.h" - класс "Капля"
///Используется для рисования отдельной "дождевой капли"
///в заданных координатах
#pragma once
namespace Rain {
    using namespace System;
    using namespace System::Drawing;
    using namespace System::Drawing::Drawing2D;
    using namespace System::Threading;
    ref class Drop {
    private:
        //Два цвета, задающих вид капли
        Color DarkColor;
        Color LightColor;
        //Размеры капли по горизонтали и вертикали
        int sizeX;
        int sizeY;
        //Позиция капли
        float x, y;
    public:
        //Свойства, позволяющие получить позицию капли
        property int SizeX   {
            int get(){
                return sizeX;
            }
        }
        property int SizeY   {
            int get(){
                return sizeY;
            }
        }
    public:
        Drop();
        //Метод, позволяющий вывести каплю,
        //одновременно сместив ее
        //на расстояние (dx, dy) относительно прежнего положения
        virtual void Paint(Graphics^ g,
                           float dx,
                           float dy);
        //Установить начальное положение капли
        virtual void Set(int x, int y);
    };
}
```

```
//Файл "Drop.cpp" - класс "Капля"
#include "Drop.h"
using namespace Rain;
Drop::Drop()
{
	Random r;//Объект, формирующий случайные числа
	x = y = -1;//Начальная позиция капли
	//Размеры капли по ширине и высоте
	sizeX = 20;
	sizeY = 40;
	//Задание светлого цвета
	LightColor = Color::FromArgb(
						100, //прозрачность (0-255)
						r.Next()%127+127,//красный
						r.Next()%127+127,//зеленый
						r.Next()%127+127 //синий
						);
	//Задание темного цвета
	DarkColor = Color::FromArgb(200,
						r.Next()%127,
						r.Next()%127,
						r.Next()%127);
}
//Рисует каплю в указанных координатах на поверхности g и
//стирает ее в прежней позиции
//Параметр dx позволяет нарисовать каплю со сдвигом на dx по оси x
//Параметр dy позволяет нарисовать каплю со сдвигом на dy по оси y
void Drop::Paint(Graphics ^g, float dx, float dy)
{
	//Получаем прямоугольную область, описанную вокруг "капли"
	Rectangle odropR(x-1, y-1, sizeX+2, sizeY+2);
	//Задаем белую сплошную кисть
	SolidBrush ^sb = gcnew SolidBrush(Color::White);
	try
	{
		//если капля была нарисована ранее, стираем ее
		if (x!=-1 && y!=-1)
		{
			//Блокируем доступ к объекту g для всех
			//потоков, кроме данного
			Monitor::Enter(g);
			try {
				g->FillRectangle(sb, odropR);
```

```
                } finally {
                    //разблокировка доступа
                    Monitor::Exit(g);
                }
            }
        }
        catch (...)
        {
        }
        //смещаем каплю
        x += dx;
        y += dy;
        //прорисовываем каплю на новом месте
        Rectangle dropR(x, y, sizeX, sizeY);
        //создаем градиентную кисть
        LinearGradientBrush^ lgb =
                gcnew LinearGradientBrush(
                                dropR,
                                LightColor,
                                DarkColor, 45,
                                true);
        //формируем изображение
        array<Point> ^ pnts = gcnew array<Point>(3);
        pnts[0].X = x+sizeX/2; pnts[0].Y = y;
        pnts[1].X = x+sizeX; pnts[1].Y = y+sizeY/2;
        pnts[2].X = x; pnts[2].Y = y+sizeY/2;
        try {
            //рисуем, предварительно получив монопольный доступ к
            //графической поверхности
            Monitor::Enter(g);
            try {
                g->FillPolygon(lgb, pnts);
                g->FillPie(lgb,
                        (int)x, (int)(y+sizeY/4),
                        sizeX, sizeY/2,
                        0, 180);
            } finally {
                Monitor::Exit(g);
            }
        } catch (...)
        {
        }
}
```

```
//Установить начальную позицию капли
void Drop::Set(int x, int y){
    this->x = (float)x;
    this->y = (float)y;
}
```

```
///Файл "FallDrop.h" - класс "Падающая капля"
#pragma once
#include "drop.h"
namespace Rain {
    using namespace System::Threading;
    ref class FallDrop : public Drop
    {
    public:
        //переопределенный метод рисования падающей капли,
        //начиная с заданных координат
        virtual void Paint(Graphics^ g,
                           float x, float y) override;
        //перегрузка метода с одним параметром для вызова из потока
        virtual void Paint(Object^ obj);
    };
    //Структура параметров, используемая для передачи данных
    //в метод потока
    value struct Param
    {
        Graphics^ g;
        float x;
        Param(Graphics^ g, float x){
            this->g = g;
            this->x = x;
        }
    };
}
```

```
//Файл "FallDrop.cpp" - класс "Падающая капля"
#include "FallDrop.h"
using namespace Rain;
void FallDrop::Paint(Graphics^ g, float x, float y){
    try {
        //Получение размеров видимой области
        //графической поверхности в монопольном режиме
        Monitor::Enter(g);
```

```
        try
        {
            RectangleF rect = g->VisibleClipBounds;
        }
        finally
        {
            Monitor::Exit(g);
        }
        //Устанавливаем каплю в начальную позицию
        Drop::Set(x, -SizeY);
        Random rnd;
        //Задаем горизонтальную составляющую скорости
        float f = rnd.NextDouble()/4;
        //Осуществляем циклическое смещение капли вниз
        for (int i = -SizeY; i <=rect.Height+1; ++i){
            Drop::Paint(g, f, 1.);
            Thread::Sleep(10);
        }
    }
    catch (...){
    }
}
//Метод преобразует параметр типа Object^
//к параметрам перегруженного метода Paint (см. выше) и вызывает его
void FallDrop::Paint(Object^ obj){
    try{
        Param ^p = (Param^)obj;
        Paint(p->g, p->x, 0);
    } catch (...){}
}
```

```
///Файл "FallRain.h" - класс "Дождь"
//В задачу класса входит создание ряда падающих капель
#pragma once
#include "FallDrop.h"
namespace Rain
{
    ref class FallRain {
    private:
        Graphics^ g;//Графическая поверхность для вывода
        bool start;
        void Do();
        Thread^ t;
```

```
    public:
        FallRain(Graphics^ g);
        //запускает "дождь"
        virtual void Start();
        //останавливает "дождь"
        virtual void Stop();
        virtual ~FallRain(void);
    };
}
```

```
///Файл "FallRain.cpp" - класс "Дождь"
#include "FallRain.h"
using namespace Rain;
FallRain::FallRain(Graphics^ g)
{
    start = false;
    //Устанавливаем режим сглаживания для
    //качественного вывода изображений
    g->SmoothingMode = SmoothingMode::HighQuality;
    this->g = g;
}
void FallRain::Start(){
    start = true;  //Устанавливаем флаг в состояние "запуск"
    //формируем поток, указывая метод для выполнения
    ThreadStart ^ts = gcnew ThreadStart(
                this,  //объект, выполняющий метод
                &FallRain::Do //ссылка на выполняемый метод
                );
    t = gcnew Thread(ts);
    //Запускаем поток
    t->Start();
}
void FallRain::Stop()
{
    //Останавливаем формирования новых капель
    start = false;
}
void FallRain::Do(){
    while (start)
    {
        //Создаем объект "падающая капля"
        FallDrop^ fd = gcnew FallDrop();
        Random r;
```

```
            try {
                //Блокируем графическую поверхность
                // для монопольного доступа
                Monitor::Enter(g);
                //Получаем размеры видимой области изображения
                try {
                    int w =
                        (int)g->VisibleClipBounds.Width -
                        fd->SizeX;
                } finally {
                    Monitor::Exit(g);
                }
                //Задаем начальную координату по оси x для капли
                int x = r.Next()%w;
                //Падение капли выполняется также в отдельном
                //потоке. Но поскольку для метода ее отображения
                // требуются параметры, создаем объект
                //параметризованного запуска
                ParameterizedThreadStart^ pts = gcnew
                    ParameterizedThreadStart(
                                fd,&FallDrop::Paint);
                //Создаем поток
                Thread^ t = gcnew Thread(pts);
                //Устанавливаем статус потока - "фоновый".
                //Все такие потоки  автоматически завершатся
                //при завершении основного потока
                t->IsBackground = true;
                //Пакуем передаваемые параметры в структуру
                Param ^p = gcnew Param(g, (float)x);
                //Запускаем поток с указанием параметров
                t->Start(p);
                //Пауза до появления следующей капли
                Thread::Sleep(500);
            } catch (...){
                break;
            }
        }
}
FallRain::~FallRain(void){
    Stop(); //При разрушении объекта останавливаем "дождь"
}
```

В. Задания для самостоятельного выполнения

1. Создать программу для вывода произвольной анимации с использованием отдельного потока.
2. Создать класс, находящий значение определителя некоторой матрицы произвольного (достаточно большого) порядка любым методом, в отдельном потоке. В процессе вычислений генерировать события, для осуществления возможности отображения прогресса вычислений в элементе **`ProgressBar`**, расположенном в окне.
3. Создать программу, способную плавно перемещать по графической поверхности (в отдельном потоке) нарисованный объект до точки, указанной щелчком мыши.

IV. Использование параллельных вычислений для решения математических задач с использованием классов библиотеки .NET

А. Теоретическая справка

Параллельный алгоритм, в отличие от традиционных, последовательных алгоритмов, может быть реализован по частям на множестве различных вычислительных устройств с последующим объединением частичных результатов в обдин общий.

Параллельные алгоритмы стали весьма актуальными благодаря постоянному совершенствованию многопроцессорных систем и увеличения числа ядер в современных процессорах. Использование нескольких вычислительных устройств одновременно позволяет во многих случаях повысить скорость вычислений по сравнению с выполнением той же задачи на одном процессоре. Однако следует иметь в виду, что некоторые задачи легко поддаются распараллеливанию, а некоторые — нет.

Так, легко распараллелить такие задачи, как, например, поиск суммы и произведения большого количества чисел, умножения двух матриц, определения простых чисел из определенного диапазона. В таких задачах итерации являются независимыми от других. Однако решение многих задач не поддается распараллеливанию или требуют применения определенных методик. В таких задачах на каждой итерации циклов требуются результаты, полученные на предыдущих этапах. В качестве примеров здесь можно привести многие алгоритмы поиска значения числа π, численные методы решения нелинейных уравнений, некоторые рекурсивные алгоритмы и.т.п.

При разработке параллельных алгоритмов требуется каким-то образом организовать связь между различными потоками, что необходимо, по крайней мере, для получения общего результата. Для этого можно воспользоваться одним из двух вариантов обмена данными: общей памятью или системой передачи сообщений.

В системах передачи сообщений применяются так называемые каналы и блоки сообщений. Системы с общей памятью требуют введения дополнительных блокировок для обрабатываемых данных. В примерах, приведенных ниже, будет разобран случай использования общей памяти.

Следует учесть, что в некоторых ситуациях, из-за дополнительной нагрузки, оказываемой на систему, параллельный код может выполняться медленнее, чем последовательный. Например, такое может произойти, когда цикл за каждую итерацию выполняет небольшой объем работ или содержит небольшое число итераций.

При создании параллельного алгоритма также требуется осуществлять балансировку нагрузки, то есть распределение процесса выполнения заданий между процессорами с целью оптимизации использования ресурсов и сокращения времени вычисления.

Для решения этих задач в среде .NET применяется библиотека параллельных задач (TPL), которая представляет собой набор открытых типов и API-интерфейсов в пространствах имен **`System::Threading`** и **`System::Threading::Tasks`**. Библиотека параллельных задач предназначена для повышения эффективности написания программ за счет упрощения процесса добавления параллелизма в приложения. Она динамически масштабирует степень параллелизма для наиболее эффективного использования всех доступных процессоров.

В качестве примера мы рассмотрим некоторые методы класса **`Parallel`**, входящего в пространство имен **`System::Threading::Tasks`**. Этот класс содержит ряд статических методов для реализации параллельно выполняющихся циклов или параллельно выполняющихся независимых задач. Рассмотрим некоторые из них подробнее.

ТАБЛИЦА IV-1. КЛАСС PARALLEL

Методы	Описания
static ParallelLoopResult For(long long fromInclusive, long long toExclusive, ParallelOptions^ parallelOptions, Action<long long>^ body)	Выполняет цикл for, обеспечивая возможность параллельного выполнения итераций и настройки параметров цикла. Рассмотрим параметры метода. *fromInclusive:* начальный индекс (включительно); *toExclusive:* конечный индекс, не включительно; *parallelOptions:* объект, используемый для настройки поведения этой операции;

Методы	Описания
	body: делегат, вызываемый один раз за итерацию. Наиболее интересным для нас в классе ParallelOptions является свойство MaxDegreeOfParallelism, позволяющее задавать максимальную степень параллелизма (число одновременно выполняющихся потоков).
generic <typename TSource> static ParallelLoopResult ForEach(IEnumerable<TSource>^ source, ParallelOptions^ parallelOptions, Action<TSource>^ body)	Выполняет операцию for each, обеспечивая возможность параллельного выполнения итераций и настройки параметров цикла. TSource — тип данных в источнике. Параметры метода: *source:* перечислимый источник данных. *parallelOptions:* объект, используемый для настройки поведения этой операции. *body:* делегат, вызываемый один раз за итерацию.
static void Invoke(ParallelOptions^ parallelOptions, ... array<Action^>^ actions)	Этот метод может использоваться для потенциально параллельного выполнения набора операций. Порядок выполнения операция при этом непредсказуем. Также нет гарантии, что те или иные операции выполняются параллельно. Этот метод не осуществляет возврат до завершения каждой из операций.

Б. Примеры

Следующие программы находят сумму:

$$\sum_{i=1}^{N} \frac{i}{i+1}, \quad N \in \mathbb{N},$$

применяя последовательный и параллельный алгоритмы.

Ниже приведена программа нахождения суммы из 2^{24} слагаемых стандартным (последовательным) образом.

```
//Файл "Summator.h"
#pragma once

ref class Summator
{
private:
    int N;
    double summ;
public:
    Summator(void);
    //Метод суммирования
    void Summ();
    //Свойство, возвращающее результат
    property double Result
    {
        double get()
        {
            return summ;
        }
    }
};
```

```
//Файл "Summator.cpp"
#include "Summator.h"
using namespace System::Threading::Tasks;
using namespace System::Threading;
Summator::Summator(void)
{
    N = 16777216; //Число слагаемых
    summ = 0;//Начальное значение суммы
}
//Метод расчета суммы последовательно
void Summator::Summ()
{
    for (double i = 1; i<=N; i++){
        summ += i/(i+1);
    }
}
```

```
//Точка входа в программу - main.cpp
#using <System.dll>
#include "summator.h"
using namespace System;
using namespace System::Diagnostics;

int main(){
    //Объект для получения суммы
    Summator^ s = gcnew Summator();
    //Таймер. С его помощью будем измерять время вычислений
    Stopwatch^ sw = gcnew Stopwatch();
    sw->Start();//Запускаем таймер
    s->Summ();//Считаем сумму из N слагаемых последовательно
    sw->Stop();//Останавливаем таймер
    //Выводим время вычислений
    Console::WriteLine("Последовательные вычисления "\
            "заняли {0} секунд",
            (double)sw->ElapsedMilliseconds/1000);
    //Выводим результат
    Console::WriteLine(s->Result);
    //Пауза, для просмотра результата перед закрытием окна
    System::Threading::Thread::Sleep(10000);
}
```

Внесем в программу изменения, позволяющие разбить процесс нахождения общей суммы на несколько частей, каждая из которых может вычисляться параллельно с другими. Для этих целей воспользуемся статическим методом For класса **`Parallel`**, который находится в пространстве имен **`System::Threading::Tasks`**.

```
//Файл "Summator.h"
#pragma once
ref class Summator {
private:
    int N;
    int parts;
    //Вокруг вещественного значащего типа
    //необходимо сделать оболочку
    //ссылочного типа для обеспечения возможности
    //синхронизации потоков
    ref struct Sum{
        double value;
    } ^summ;
```

```
    //Закрытый метод, вычисляющий частичные суммы
    void Summ(int part);
public:
    Summator(void);
    //Общедоступный метод для начала параллельных вычислений
    void Summ();
    property double Result{
        double get(){
            return summ->value;
        }
    }
};
```

```
//Файл "Summator.cpp"
#include "Summator.h"
using namespace System::Threading::Tasks;
using namespace System::Threading;
Summator::Summator(void)
{
    N = 16777216;//Число слагаемых
    //Выделяем память для ссылочной «оболочки» результата
    summ = gcnew Sum();
    //Устанавливаем начальное значение
    summ->value = 0;
    //Задаем количество частей, на которые делится
    // весь объем вычислений
    parts = 8;
}

//Запускаем вычислительный процесс в параллельном режиме
void Summator::Summ()
{
    //В качестве действия, выполняемого циклом,
    //используется перегруженный метод Summ
    //с целочисленным параметром
    Parallel::For(
        0,//Номер первой части слагаемых (начальный индекс).
        parts,  //Конечный индекс, не включаемый в интервал.
        //Выполняемое действие, куда цикл передает номер итерации:
        gcnew System::Action<int>(this,&Summator::Summ)
                );
}
```

```
//Метод, вычисляющий частичные суммы.
//Вызывается из цикла Parallel::For, находящегося
//в перегруженном методе Summ
void Summator::Summ(int part){
    //Количество элементов в одной части
    int partSize = (int)(N/parts);
    //Остаток элементов
    //(которые нельзя поровну разделить между частями)
    int ost = N - partSize*parts;
    //Стартовый индекс слагаемого
    int st = (part*partSize) +
                ((part < ost)?part:ost) + 1;
    //Конечный индекс слагаемого
    int fin = ((part+1)*partSize) +
              (((part+1) < ost)?(part+1):ost);
    //Сумма, вычисляемая данным потоком
    double s = 0;
    for (double i = st; i<=fin; i++){
        s += i/(i+1);
    }
    //Добавляем полученный результат к общей сумме.
    //При доступе к общему объекту необходимо синхронизировать
    //работу потоков, чтобы не произошло собев в программе.
    Monitor::Enter(summ);//Блокируем ссылочный объект summ
        //В этой секции одновременно с объектом summ
        //может работать только один поток!
        try {
            summ->value += s;
        } finally {
            //Освобождаем объект summ для других потоков
            Monitor::Exit(summ);
        }
}
```

```
//Точка входа в программу - main.cpp
#using <System.dll>
#include "summator.h"
using namespace System;
using namespace System::Diagnostics;
int main()
{
    //Объект для вычисления суммы
    Summator^ s = gcnew Summator();
```

```
    //Таймер. С его помощью будем измерять время вычислений
    Stopwatch^ sw = gcnew Stopwatch();
    sw->Start();//Запускаем таймер
    //Запускаем вычислительный процесс в режиме параллелизма:
    s->Summ();
    sw->Stop();//Останавливаем таймер
    Console::WriteLine("Параллельные вычисления "\
            "заняли {0} секунд",
            (double)sw->ElapsedMilliseconds/1000);
    Console::WriteLine(s->Result); //Выводим результат
    //Пауза, для просмотра результата перед закрытием окна
    System::Threading::Thread::Sleep(10000);
}
```

Время работы последовательного и параллельного алгоритма может отличаться в разы в зависимости, от количества ядер процессора на конкретном компьютере. Однако ускорение не будет пропорционально количеству процессоров, установленных в системе, и, как правило, будет несколько ниже. Например, на компьютере с четырехядерном процессором Intel® Core™2 Quad Q9300 @ 2,50Ггц последовательный способ вычислений занял по времени 0,137с., а параллельный — 0,054 секунды, что более чем в 2,5 раза быстрее.

В. Задания для самостоятельного выполнения

1. Написать 2 класса, для работы с матрицами в режиме последовательной и параллельной обработки. Реализовать операции умножения матрицы на число, сложения двух матриц, умножения матриц. Сравнить время вычислений при использовании этих классов.
2. Написать классы для вычисления интегралов при помощи приближенных методов (левых прямоугольников, средних прямоугольников, трапеции) с использованием последовательных и параллельных алгоритмов. Сравнить время вычислений при использовании этих классов.
3. Написать класс, генерирующий изображение фрактала и выводящий его на экран. Изображение формировать двумя способами: последовательно и параллельно. Сравнить время построения изображения при использовании каждого из этих способов.

4. Написать классы для осуществления сортировки массивов применяя последовательные и параллельные варианты одинаковых алгоритмов. Сравнить время их выполнения.
5. Написать класс для поиска всех простых чисел в заданном диапазоне и получения упорядоченного по возрастанию результата в файле. Реализовать в классе методы, использующие последовательный и параллельный алгоритм поиска. Определить время решения задачи в каждом случае.

Литература

а) Основная литература

1. Шеферд Дж. Программирование на Microsoft Visual C++ .NET. — СПб., «Питер», «Русская редакция», 2007. — 928с.
2. Корера А., Фрейзер С., Маклин С. и др. Visual C++ .NET. Пособие для разработчиков C++. — М.: «Лори», 2003. — 416с.
3. Медведев В.И. Особенности объектно-ориентированного программирования на C++/CLI, C# и Java, М.: РИЦ «Школа», 2010г. – 444с.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы, построение и анализ. — М. ЦНМО, 2000.
5. Павловская Т.А., Щупак Ю.А. C/C++. Структурное программирование: практикум. – СПб.: «Питер», 2007г. – 239с.
6. Павловская Т.А., Щупак Ю.А. C++. Объектно-ориентированное программирование: практикум. – СПб.: «Питер», 2008г. – 265с.
7. Хогенсон Г. C++/CLI: язык Visual C++ для среды .NET., М.: «Вильямс», 2007г. – 464 с.
8. Хабибуллин И. Ш. Программирование на языке высокого уровня. C/C++. — СПб.: BHV-Петербург, 2006. — 512 с.
9. Шилдт Г. Теория и практика C++: пер. с англ. — СПб.: BHV-Санкт-Петербург, 1996. — 416 с. ил.

б) Дополнительная литература:

10. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. — М.: ДМК-Пресс, 2011. — 272 с.
11. Лаптев В.В., Морозов А.В., Бокова А.В. C++. Объектно-ориентированное программирование. Задачи и упражнения. – СПб.: Питер, 2007г. – 288 с.
12. Шень А. Программирование: теоремы и задачи. — М.: МЦНМО, 1995.
13. Кнут Д. Искусство программирования для ЭВМ. — В 3-х т. — М.: Мир, 2000.
14. Страуструп Б. Язык программирования C++, М.: "Бином", 1999.
15. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: "Бином", 1999.

16. Пол А. Объектно-ориентированное программирование на C++ М.: "Бином", 1999.

в) Программное обеспечение и Интернет-ресурсы

15. http://tulpar.kfu-elearning.ru/course/view.php?id=81
16. http://citforum.ru/
17. http://www.emanual.ru/
18. http://rsdn.ru
19. http://msdn.microsoft.com/ru-ru/
20. http://intuit.ru/

Сергей Владиславович Маклецов

Компьютерный практикум (часть 2)

Учебно-методическое пособие

Отпечатано в полном соответствии с предоставленым оригинал-макетом

Подписано в печать 05.02.2013 г.
Форм. 60x84 1/16. Гарнитура Times. Печать ризографическая.
Печ. л. 6,75. Тираж 100. Заказ 25.

Лаборатория оперативной полиграфии издательства КФУ

Адрес: 420045, Казань, Кр. Позиция, 2а. Тел. 233-72-12