Системное программирование

Урок № 3

Небезопасный код, управление памятью, использование реестра, создание dll модулей

Содержание

1. **Небезопасный код** 3
2. **Особенности управления памятью при разработке приложений платформы Microsoft .Net** 8
3. **Основные сведения о реестре** 20
4. **Работа с реестром с помощью WinAPI** 31
5. **Работа с реестром с помощью платформы Microsoft .Net** 67
6. **Разработка динамически подключаемых библиотек с использованием .Net Framework** 75
7. **Создание хуков в приложениях платформы Microsoft .Net** 91
8. **Домашнее задание** 104

1. Небезопасный код

Думаю, название темы у вас вызвало как минимум удивление, как это код может быть небезопасным, на самом деле небезопасный он только с точки зрения идеологии программирования .NET и зачастую означает лишь то, что в вашей программе используются указатели.

Технология .NET позволяет использовать так называемый небезопасный (unsafe) код в приложениях – код, который не контролируется системой Common Language Runtime (CLR). Кроме этого, unsafe код не подчиняется средствам управления и ограничениям, налагаемым на обычный, управляемый код. Зачем же нужен такой код, который может привести к нестабильной работе или вообще к полной неработоспособности приложения? Все дело в том, unsafe код позволяет использовать ресурсы, которые запрещены в безопасном коде. А использование таких ресурсов, может сделать ваше приложение более эффективным и расширить его возможности. Однако вы должны понимать, что применение небезопасного кода требует от программиста аккуратности и ответственного отношения. Что же это за ресурсы, которыми позволяет пользоваться unsafe код?

Из курса программирования C++ вы должны помнить, что такое указатель, но на всякий случай напомним.

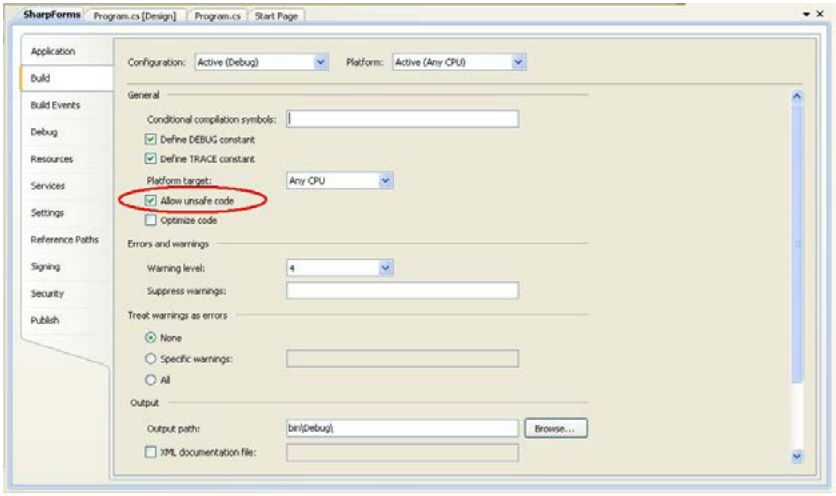
Указатель – это переменная содержащая адрес другой переменной.

Можно предположить, что указатель это некого рода подобие ссылок в C#. На самом деле это не совсем так. Ос-

новное отличие между ними заключается в том, что указатель может указывать на что угодно в памяти, а ссылка всегда указывает на объект своего типа. И естественно из этого следует, что если указатель может указывать на что угодно, возможно неправильное его использование, которое приведет к ошибкам в памяти, которые к тому же будет очень сложно отыскать. Вот почему работа с указателями не поддерживается в управляемом (контролируемом) коде, и все операции с указателями должны быть отмечены как небезопасные (unsafe).

Для использования небезопасного кода в C# необходимо приложение скомпилировать с параметром компилятора **/unsafe**. Для этого нужно проделать следующие действия:

- открыть свойства проекта (Project->Properties) (Alt+F7);
- выбрать закладку Build;
- установить флажок Allow unsafe code.



Или если вы используете, компилятор из командной строки просто скомпилируйте код с ключом /unsafe.

Пример:

```
csc /unsafe file.cs
```

Как уже говорилось, ключевое слово unsafe разрешает использовать небезопасный код и, в частности, позволяет работать с указателями

Модификатор unsafe можно использовать различными способами. Первый – это возможность использования его при объявлении метода, класса, структуры, делегата и таким образом вся часть кода соответствующей декларации становится небезопасной, и в нем, соответственно, становится возможным использовать указатели.

Пример:

```
//Область небезопасного кода начинается от списка
//параметров
unsafe public void MyUnsafeMethod(int* pFirstParam,
char* pSecondParam)
{
     //Здесь находится небезопасный код
}//Здесь заканчивается область небезопасного кода

//Небезопасная структура,
//внутри которой обьявлены небезопасные поля
public unsafe struct MyStruct
{
     public int Val1;
     public unsafe char* Val2;
     public unsafe MyStruct* Val3;
}
```

```
//Объявление небезопасного класса
public unsafe class A
{
    ...
}
```

Так же можно использовать так называемый небезопасный блок. Небезопасный блок – это часть кода, помеченная флагом unsafe, внутри которого можно использовать небезопасный код.

Пример:

```
//Начало небезопасного блока
unsafe
{
    // Здесь можно использовать небезопасный код
}//Конец небезопасного блока
```

Рассмотрим небольшой пример, создадим класс и в нем два метода, в которых будет использоваться небезопасный код.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace UnsafeCode
{
    class Program
    {
        //Метод является небезопасным
        //возводит в степень y число x
        unsafe static void Pow(int* x, int y)
        {
            int z = *x;
            for (int i = 1; i < y; i++)
                *x *= z;
        }
```

```
        unsafe static void Main(string[] args)
        {
            int param = 2;
            //Вызов небезопасного метода
            Pow(&param, 4);
            Console.WriteLine("value is {0}",param);
        }
    }
}
```

Следует сказать, что вместо указателей, конечно можно было бы использовать и ссылки. Тем более, что практически каждый тип указателя который использовался в контексте C или C++, заменен ссылкой в C#. Однако существуют ситуации, когда использование указателя и соответственно небезопасного кода становится необходимостью. Например, прямое взаимодействие с операционной системой, прямой доступ к памяти устройства, или реализация алгоритма с критическим временем исполнения (т.н. time-critical algorithm).

2. Особенности управления памятью при разработке приложений платформы Microsoft .Net

Главной целью платформы **Microsoft .Net** является возможность сделать процесс программирования более простым и производительным. Надежность управления памятью обеспечивается "проповедованием" безопасного программирования.

Для этого среда исполнения **.Net** предоставляет механизмы управления памятью, например, сборщик мусора (**garbage collector**, **GC**). При написании небезопасного кода необходимо очищать за собой память самостоятельно, а при использовании безопасного кода, на это, в большинстве случаев, можно не обращать внимания. Однако сборщик мусора считать панацеей не следует.

Общий принцип управления памятью в приложениях платформы **Microsoft.Net** формулируется очень просто: для создания объекта в области контролируемой кучи (**managed heap**) используется ключевое слово **new**. Менеджер памяти распределяет всю память, занятую объектами, в непрерывном блоке в начале кучи и удовлетворяет запрос на создание нового объекта, выделяя область, непосредственно следующую за уже распределенным участком.

Выделение памяти выполняется очень эффективно, но доступная память не бесконечна. Кроме того, в процессе работе в куче образуются дыры (отработавшие объекты становятся ненужными). В этом случае запускается сборщик мусора, который перемещает "рабочие" объекты в начало кучи – достигается непрерывность кучи.

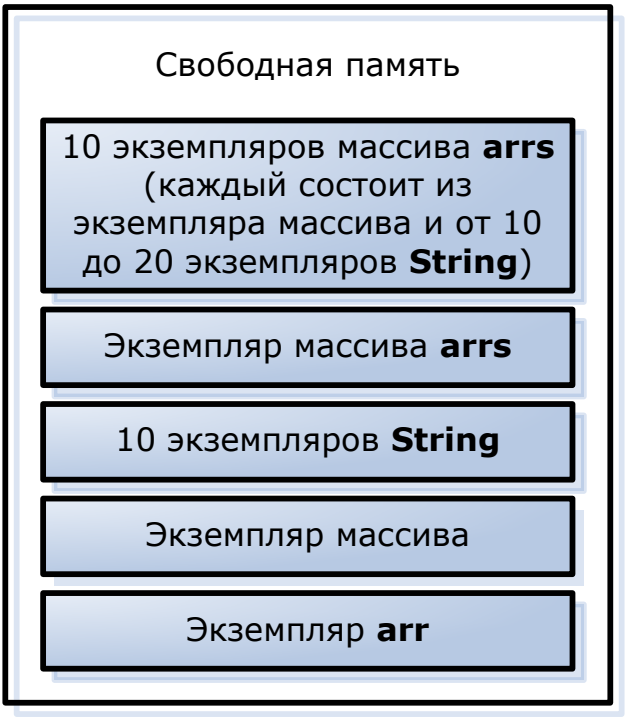
Следующая программа демонстрирует, как работает механизм выделения памяти.

```
using System;

namespace ControlMemory
{
    class DemoNew
    {
        int max;
        String[] AnArray;
        public DemoNew()
        {
            max = 10;
            AnArray = new String[max];
        }
        public DemoNew(int max)
        {
            this.max = max;
            AnArray = new String[max];
        }
        public String this[int i]
        {
            get
            {
                if (i < 0 || i >= max) return null;
                else return AnArray[i];
            }
            set
            {
                if (i >= 0 && i < max) AnArray[i] =
                                                value;
            }
        }
```

```
        public int Count
        {
            get { return max; }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // создание объекта
            DemoNew arr = new DemoNew();
            Console.WriteLine(
                      "Введите элементы объекта arr
                       {0})",arr.Count);
            for (int i = 0; i < arr.Count; i++)
            {
                Console.Write("{0}: ",i);
                arr[i] = Console.ReadLine();
            }
            int num;
            do {
                Console.WriteLine(
                      "Введите номер элемента (от 0 до
                      {0}):",arr.Count - 1);
                num = Convert.ToInt16(Console.Read
                                              Line());
            } while (num < 0 || num >= arr.Count);
            Console.WriteLine(arr[num]);
            // создание массива объекта
            DemoNew[] arrs = new DemoNew[10];
            Random K = new Random();
            for (int i = 0; i < 10; i++)
                arrs[i] = new DemoNew(K.Next(10,20));
            foreach (DemoNew d in arrs)
                Console.Write("{0}\t", d.Count);
            // потеря ссылок
            arr = arrs[3];
            // дальнейшая работа
        }
    }
}
```

При создании экземпляра типа **DemoNew**, память выделяется в начале свободной области в куче. В конструкторе выделяется память для массива и объектов **String** непосредственно после объекта **arr**.

После создания массива объектов **DemoNew** выделяется память для массива, его экземпляров и объектов каждого экземпляра. В результате получим картину, изображенную на рисунке.



В последней строке программы переменной **arr** присваивается ссылка на один из экземпляров массива **arrs**. Таким образом, ссылка на созданный ранее объект теряется, и в памяти появляются потерянные ссылки или,

другими словами, мусор. Ни на исходный объект, ни на массив строк, ни на сами содержащиеся в нем строки ссылок больше не существует. Воспользоваться этими объектами нет никакой возможности, поэтому их нужно уничтожить и освободить память. Данная ситуация изображена на следующем рисунке.

Если все время только создавать все новые и новые объекты, то память, в конце концов, исчерпается. В этом случае будет сгенерировано исключение **OutOfMemoryException**. Поэтому, для исключения ошибок, необходимо поступить следующим образом:

```
try
{
    DemoNew demo = new DemoNew(36);
}
catch (OutOfMemoryException e)
{
    Console.WriteLine(e.Message);
}
```

После этого, автоматически запускается сборщик мусора, который строит граф ссылок, начиная с каждого корня. Поскольку в каждом типе имеются метаданные, он помещает каждый корень в список, затем добавляет объекты, на которые ссылаются корни и рекурсивно их все посещает. По завершении операции объекты, которые программа уже не может использовать, уничтожаются на следующем шаге, когда сборщик начнет обходить и уплотнять кучу. По завершении процедуры все оставшиеся объекты снова занимают непрерывную область.

Эта полностью работающая автоматически схема управления памятью имеет как положительные, так и отрицательные стороны: это упрощает процесс программирования; процесс удаления объектов (закрытия соединения с базой данных, окна **Windows** и т.п.) происходит в соответствии с неизвестным алгоритмом.

Для того чтобы обеспечить удаление объектов из памяти в соответствии с необходимыми правилами, нужно реализовать метод **System.Object.Finalize()**. В C# запрещено напрямую замещать этот метод, а также вызывать напрямую. Для этого необходимо использовать в определении класса метод, очень похожий на деструктор C++:

```
class DemoNew
{
     ...
     ~DemoNew()
     {
          Console.WriteLine("Освобождение ресурсов");
     }
}
```

При размещении объекта в куче при помощи оператора **new** автоматически определяется, поддерживает ли объект метод **Finalize()**. Если этот метод поддерживается, ссылка на объект помечается как завершаемая (**finalizable**). При удалении объекта из памяти, сборщик мусора запускает деструктор, прежде чем будет произведено физическое удаление объекта из памяти.

В случаях, когда существуют ценные ресурсы, за которые, например, нужно платить и хотелось бы как можно скорее освободить, использование пользовательских деструкторов не особо подходит, поскольку они вызываются в соответствии с расписанием работы сборщика мусора.

В этом случае рекомендуется использовать интерфейс **IDisposable**, который содержит один единственный метод **Dispose()** (освободить):

```
public interface IDisposable
{
    public void Dispose();
}
```

Метод **Dispose()** пользователь может вызвать вручную, сразу после использования объекта и до того, как объект выйдет за пределы области видимости. Таким

образом, можно гарантировать освобождение ресурсов без помещения указателя на деструктор в очередь завершения.

Например, применить интерфейс в случае класса **DemoNew** можно следующим образом:

```
class DemoNew : IDisposable
{
    ...
    public void Dispose()
    {
        Console.WriteLine("Освобождение ресурсов");
    }
}
```

Используя такой подход, пользователь может в любое время освободить наиболее ценные ресурсы, не загружая дополнительно очередь завершения. Кроме того, можно сочетать применение интерфейса **IDisposable** и пользовательского деструктора.

Сборщик мусора – это объект, к которому можно обращаться через ссылку. Для работы со сборщиком мусора предназначен специальный класс **System.GC**, от которого нельзя производить другие классы. В нем определен набор статических членов, при помощи которых можно осуществить взаимодействие со сборщиком мусора:

Член	Назначение
`Collect()`	Заставляет сборщик мусора заняться выполнением своих обязанностей для всех поколений. По желанию можно указать в качестве параметра конкретное поколение.
`GetGeneration()`	Возвращает поколение, к которому относится данный объект.
`MaxGeneration`	Возвращает максимальное количество поколений, поддерживаемое данной системой.
`ReRegisterForFinalize()`	Устанавливает флаг возможности завершения для объектов, которые ранее были помечены как незавершаемые при помощи метода SupressFinalize().
`SuppressFinalize()`	Устанавливает флаг запрещения завершения для объектов, которые в противном случае могли бы быть завершены сборщиком мусора.
`GetTotalMemory()`	Возвращает количество памяти (в байтах), которое в настоящее время занимают объекты в управляемой куче, включая те объекты, которые будут вскоре удалены. Этот метод принимает параметр типа boolean, с помощью которого можно указать, запускать или нет процесс сборки мусора при вызове этого метода.

Ниже приведен фрагмент программы, иллюстрирующий взаимодействие со сборщиком мусора:

```
class DemoNew : IDisposable
{
     ...
     public void Dispose()
     {
          Console.WriteLine("Освобождение ресурсов");
          // подавляем завершение
          GC.SuppressFinalize(this);
     }
     ~DemoNew()
     {
          Dispose();
     }
}
```

Этот вариант класса поддерживает как деструктор, так и интерфейс **IDisposable**. В методе **Dispose()** происходит вызов метода **GC.SupressFinalize()**, что говорит системе об отсутствии необходимости, вызывать деструктор объекта.

Как было сказано выше, в классе **System.GC** существует понятие поколения (**generation**) – концепция, предназначенная сделать процесс сборки мусора более удобным.

При пометке объектов для завершения, сборщик мусора не проверяет все подряд объекты приложения (может занять много времени). Для повышения производительности сборки мусора, все объекты в куче разбиты на поколения. **Чем дольше объект существует в куче, тем больше вероятность того, что он будет нужен и в дальнейшем**. И наоборот – недавно появившиеся объекты быстро перестанут быть нужными (например, временные объекты). В связи с этим, каждый объект относится к одному из следующих поколений:

- 0: недавно появившиеся объекты, которые еще не проверялись сборщиком мусора;
- 1: объекты, которые пережили одну проверку сборщика мусора (не удалены физически, поскольку в куче было достаточно свободного места);
- 2: объекты, которые пережили более чем одну проверку сборщика мусора.

При запуске процесса сборки мусора **System.GC** в первую очередь производит проверку и удаление всех объектов поколения 0. Если освободилось достаточно места, оставшиеся объекты поколения 0 переводятся в поколение 1, иначе запускается процесс проверки объектов поколения 1, а при необходимости и поколения 2. Таким образом, недавно созданные объекты обычно удаляются быстрее.

Для определения, к какому поколению относится объект, используется метод **GC.GetGeneration()**. Кроме того, метод **GC.Collect()** позволяет указать поколение, проверяемое при вызове сборщика мусора.

Обращаться к сборщику мусора следует только по надобности. Необдуманное вмешательство в его работу может привести к снижению производительности приложения.

В языке C# есть оператор, полезный для управления ресурсами, – **using**. Для объектов, реализующих интерфейс **IDisposable**, предложение **using** вызывает метод **Dispose()** после завершения работы с объектом. В следующем фрагменте кода (см. раздел 7) для установки хука применяются объекты типов **Process** и **ProcessModule**:

```
using (Process curProcess = Process.
                                    GetCurrentProcess())
using (ProcessModule curModule = curProcess.
                                     MainModule)
{
    return SetWindowsHookEx(WH_KEYBOARD_LL, proc,
                                GetModuleHandle(curModule.
                                          ModuleName), 0);
}
```

Код блока, следующего за **using**, может манипулировать объектами, а при выходе для каждого объекта вызывается метод **Dispose()**.

3. Основные сведения о реестре

Реестр Windows – это своеобразная иерархическая база данных, в которой хранится информация о настройках Windows, настройках установленных программ, о подключенных к компьютеру устройствах и их настройках и т. д.

Ранние версии ОС (MS DOS и Windows 1.0–3.11) хранили свои настройки в текстовых файлах **config.sys** и **autoexec.bat**, каждая программа так же имела свой собственный конфигурационный файл (**.ini**). Это было неудобно, а так же вносило изрядную путаницу, поэтому, начиная с Windows NT, в качестве хранилища настроек используется реестр.

Реестр физически хранится в различных файлах. Часть из них находятся в папке **%SystemRoot%\System32\Config**. Вот они:

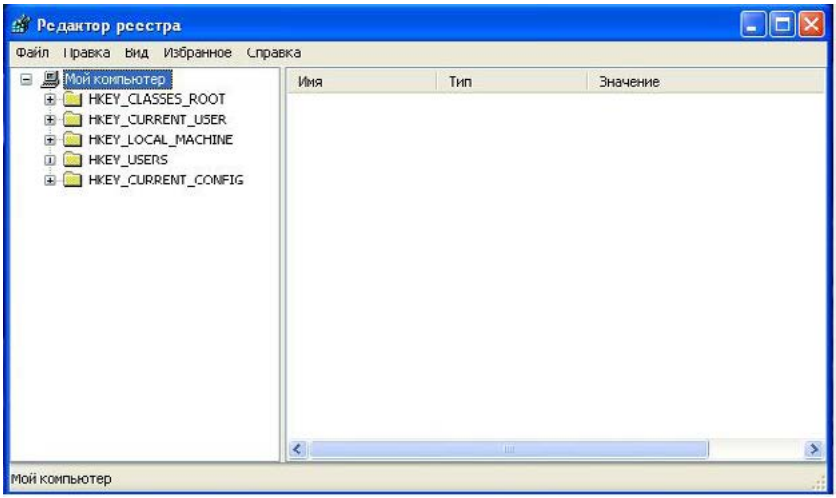
- components;
- default;
- sam;
- security;
- software;
- system.

Как видите, имена файлов без расширений. Копии этих файлов хранятся в каталоге **%SystemDrive% \Windows\Repair**. Файлы, используемые для хранений личных на-

строек пользователя, находятся в директории соответствующей учетной записи **%UserProfile%** это:

- .recently-used.xbel;
- Ntuser.dat;
- NTUSER.DAT.LOG.

Поскольку файлов в реестре несколько, и хранятся они распределено, его нельзя открыть в текстовом редакторе и внести какие-либо коррективы, для работы с ним требуется специальная программа – редактор реестра. Это системная программа и называется она regedit. Ее можно запустить с помощью команды regedit, в окне запуска программ, при нажатии на комбинацию клавиш Win+R.



Реестр отчасти напоминает файловую систему жесткого диска. Здесь присутствуют свои каталоги, в которых хранятся данные, только называются они по-другому. Каталоги в реестре называются **разделами** или

ключами (Key). Разделы могут хранить в себе вложенные разделы или **параметры**. В параметрах и хранятся все настройки ОС.

Давайте для начала разберемся с основными (корневыми) разделами реестра, их всего пять. Это:

- HKEY_CLASSES_ROOT: Содержит информацию о зарегистрированных типах файлов и объектах COM и ActiveX. Вместо полного имени раздела иногда используется сокращение HKCR;
- HKEY_CURRENT_USER: Содержит настройки текущего пользователя, Вместо полного имени раздела иногда используется аббревиатура HKCU. Такой раздел в реестре будет создаваться для каждого пользователя;
- HKEY_LOCAL_MACHINE: Содержит параметры конфигурации, относящиеся к данному компьютеру. Раздел, в отличие от HKCU, один для всех пользователей. Вместо полного имени раздела иногда используется аббревиатура HKLM;
- HKEY_USERS: Раздел содержит все активные загруженные профили пользователей ПК. Вместо полного имени раздела иногда используется аббревиатура HKU;
- HKEY_CURRENT_CONFIG: Содержит сведения о профиле оборудования, используемом локальным компьютером при запуске системы. Вместо полного имени раздела иногда используется аббревиатура HKCC.

Как уже упоминалось, в разделах могут храниться значения (**параметры**). Максимальная длина имени параметра:

- Windows Server 2003, Windows XP и Windows Vista: 16 383 символов;
- Windows 2000: 260 символов ANSI или 16 383 символа Юникод;
- Windows 95, Windows 98 и Windows Millennium Edition: 255 символов.

Значения большого размера (больше 2048 байт) хранятся во внешних файлах, а в реестр заносится имя такого файла. Это способствует повышению эффективности использования реестра. Максимальный размер параметра:

- Windows NT 4.0/Windows 2000/Windows XP/Windows Server 2003/Windows Vista: Доступная память;
- Windows 95, Windows 98 и Windows Millennium Edition: 16 300 байт.

Общий размер всех параметров раздела не должен превышать 64 КБ.

Следующая таблица содержит список типов данных, определенных и используемых Windows.

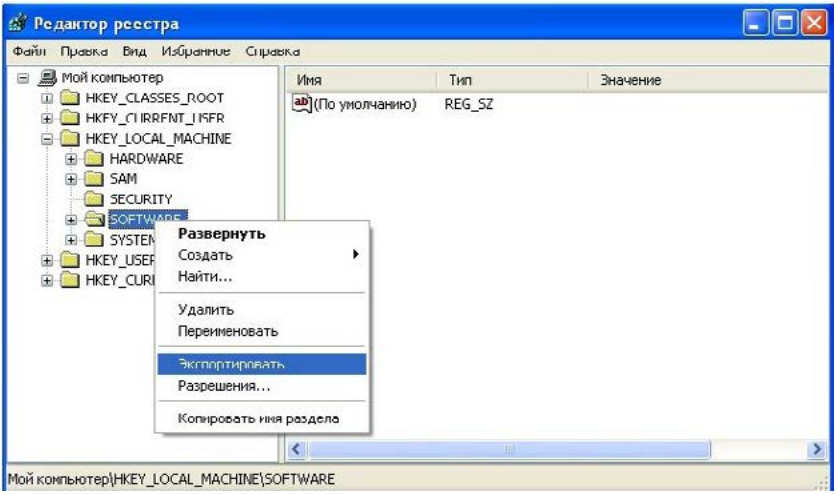
Имя	Тип	Описание
Двоичный параметр	REG_BINARY	Необработанные двоичные данные. Большинство сведений об аппаратных компонентах хранится в виде двоичных данных и выводится в редакторе реестра в шестнадцатеричном формате.

Имя	Тип	Описание
Параметр DWORD	REG_DWORD	Данные представлены в виде значения, длина которого составляет 4 байта (32-разрядное целое). Этот тип данных используется для хранения параметров драйверов устройств и служб. Значение отображается в окне редактора реестра в двоичном, шестнадцатеричном или десятичном формате. Эквивалентами типа DWORD являются DWORD_LITTLE_ENDIAN (самый младший байт хранится в памяти в первом числе) и REG_DWORD_BIG_ENDIAN (самый младший байт хранится в памяти в последнем числе).
Расширяемая строка данных	REG_EXPAND_SZ	Строка данных переменной длины. Этот тип данных включает переменные, обрабатываемые при использовании данных программой или службой.
Многострочный параметр	REG_MULTI_SZ	Многострочный текст. Этот тип, как правило, имеют списки и другие записи в формате, удобном для чтения. Записи разделяются пробелами, запятыми или другими символами.

Имя	Тип	Описание
Строковый параметр	REG_SZ	Текстовая строка фиксированной длины.
Двоичный параметр	REG_ RESOURCE_LIST	Последовательность вложенных массивов. Служит для хранения списка ресурсов, которые используются драйвером устройства или управляемым им физическим устройством. Обнаруженные данные система сохраняет в разделе \ResourceMap. В окне редактора реестра эти данные отображаются в виде двоичного параметра в шестнадцатеричном формате.
Двоичный параметр	REG_ RESOURCE_ REQUIREMENTS_ LIST	Последовательность вложенных массивов. Служит для хранения списка драйверов аппаратных ресурсов, которые могут быть использованы определенным драйвером устройства или управляемым им физическим устройством. Часть этого списка система записывает в раздел \ResourceMap. Данные определяются системой. В окне редактора реестра они отображаются в виде двоичного параметра в шестнадцатеричном формате.

Имя	Тип	Описание
Двоичный параметр	REG_FULL_RESOURCE_DESCRIPTOR	Последовательность вложенных массивов. Служит для хранения списка ресурсов, которые используются физическим устройством. Обнаруженные данные система сохраняет в разделе \HardwareDescription. В окне редактора реестра эти данные отображаются в виде двоичного параметра в шестнадцатеричном формате.
Отсутствует	REG_NONE	Данные, не имеющие определенного типа. Такие данные записываются в реестр системой или приложением. В окне редактора реестра отображаются в виде двоичного параметра в шестнадцатеричном формате.
Ссылка	REG_LINK	Символическая ссылка в формате Юникод.
Параметр QWORD	REG_QWORD	Данные, представленные в виде 64-разрядного целого. Начиная с Windows 2000, такие данные отображаются в окне редактора реестра в виде двоичного параметра.

Так как при поврежденном реестре мы получим неработоспособную либо частично неработоспособную ОС, при изменениях в реестре рекомендуется делать резервную копию (backup) ветки реестра, в которой производятся изменения, либо всего реестра. Если делать это с помощью редактора реестра regedit, то резервная копия будет сохранена в специальном формате REG, а файл резервной копии называют REG-файлом. Этот файл имеет свою структуру и синтаксис, и его можно редактировать любым текстовым редактором. Для того чтобы сделать резервную копию какого-либо раздела, достаточно в редакторе реестра нажать правой кнопкой на имени раздела и выбрать "Экспортировать", после чего выбрать место, куда будет сохранена резервная копия ветки реестра, либо выбрав ветку реестра нажать меню Файл -> Экспорт.



Давайте рассмотрим синтаксис REG-файла.

Синтаксис REG-файла:

Версия_редактора_реестра
Пустая строка
[Путь_реестра1]
"Имя_элемента_данных1"="Тип_данных1:Значение_данных1"
Имя_элемента_данных2"="Тип_данных2:Значение_данных2"
Пустая строка
[Путь_реестра2]
"Имя_элемента_данных3"="Тип_данных3:Значение_данных3"

Пример файла реестра:

Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\MyKey]
"MyStringVal"="String Val"
"MyBinaryVal"=hex:10,23,34
"MyDWORDVal"=dword:0000ac12
[HKEY_LOCAL_MACHINE\SOFTWARE\MyKey\MySubKey]
"SubKeyVal"="SubKey"

Версия_редактора_реестра — либо "Windows Registry Editor Version 5.00" для Windows 2000, Windows XP и Windows Server 2003, либо "REGEDIT4" для Windows 98 и Windows NT 4.0. Заголовок REGEDIT4 можно также использовать на компьютерах с системой Windows 2000, Windows XP и Windows Server 2003.

Пустая строка обозначает начало новой ветки реестра. Каждый раздел или подраздел является новым

путем реестра. Так же пустые строки облегчают анализ содержимого файла.

Путь_реестра_x – путь подраздела, который содержит первое импортируемое значение. Путь должен быть заключен в квадратные скобки, а каждый новый уровень в иерархии отделен обратной косой чертой.

Например:

[HKEY_LOCAL_MACHINE\SOFTWARE\Policies\ Microsoft\ Windows\System]

REG-файл может содержать несколько путей реестра. Если нижняя часть иерархии в инструкции пути отсутствует в реестре, создается подраздел. Содержимое файлов реестра передается в реестр в порядке их ввода. Поэтому для создания подраздела с вложенным подразделом, необходимо вводить строки в соответствующем порядке.

Имя_элемента_данных_x – это имя импортируемого элемента данных. Если элемент данных файла отсутствует в реестре, он будет добавлен (со значением). Если элемент данных существует, значение в REG-файле переписывает существующее значение. Имя элемента данных заключается в кавычки. За именем элемента данных следует знак равенства (=).

Тип_данных_x – это тип данных значения реестра; указывается после знака равенства. Для всех типов данных, кроме REG_SZ (строковое значение), за типом данных следует двоеточие. В случае с REG_SZ необходимо поставить знак равенства.

Значение_данных_x следует сразу же за двоеточием (или знаком равенства в случае REG_SZ) и должно

иметь надлежащий формат (например, быть строкой или шестнадцатеричным значением). Для двоичных элементов данных необходимо использовать шестнадцатеричный формат.

Для того чтобы удалить раздел реестра с помощью REG-файла, перед разделом следует поставить знак - (дефис). Например, для удаления раздела **Test** из раздела реестра **HKEY_LOCAL_MACHINE\Software** необходимо в REG-файле написать следующую строку:

[-HKEY_LOCAL_MACHINE\Software\Test]

Для удаления значения реестра с помощью REG-файла необходимо после знака равенства, который идет за именем элемента поставить минус (–). Например для удаления значения реестра **TestValue** из раздела **HKEY_LOCAL_MACHINE\Software\Test** в REG-файле необходимо написать строку:

[HKEY_LOCAL_MACHINE\Software\Test]

"TestValue"=-

4. Работа с реестром с помощью WinAPI

Итак, мы с вами разобрались с тем, что такое реестр и для чего он нужен операционной системе Windows. Теперь давайте разберемся с тем, как работать с реестром средствами WinAPI. Для этого существуют следующие функции.

RegOpenKeyEx – функция предназначена для открытия определенного ключа реестра. Следует сразу уточнить, что названия ключей регистронезависимы. Синтаксис функции выглядит следующим образом:

```
LONG WINAPI RegOpenKeyEx(
  __in       HKEY hKey,
  __in_opt   LPCTSTR lpSubKey,
  __reserved DWORD ulOptions,
  __in       REGSAM samDesired,
  __out      PHKEY phkResult
);
```

Давайте разберемся с параметрами подробнее.

hKey – входящий параметр, описатель открытого ключа реестра, этот параметр может быть возвращен функцией **RegCreateKeyEx** или **RegOpenKeyEx** или может быть одним из следующих предопределенных разделов:

HKEY_CLASSES_ROOT;

HKEY_CURRENT_USER;

HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpSubKey – входящий параметр, имя подраздела реестра (вложенной ветки реестра) который нужно открыть. Если этот параметр NULL, а hkey является HKEY_CLASSES_ROOT, phkResult получит новый описатель на ключ определенный в параметре **hKey**.

ulOptions – указывает способ, который необходимо применить при открытии ключа. Этот параметр должен быть 0.

samDesired – входящий параметр; маска – определяющая, с какими правами доступа мы хотим открыть ветку реестра. Функция завершается неудачей, если дескриптор безопасности ключа не разрешает запрашиваемый доступ для вызываемого процесса.

Существуют следующие уровни доступа к ключам реестра:

KEY_ALL_ACCESS (0xF003F) – комбинация параметров STANDARD_RIGHTS_REQUIRED, KEY_QUERY_VALUE, KEY_SET_VALUE, KEY_CREATE_SUB_KEY, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, and KEY_CREATE_LINK. Иными словами это доступ со всеми привилегиями – полный доступ к ключу реестра.

KEY_CREATE_LINK (0x0020) – зарезервирован для системного использования.

KEY_CREATE_SUB_KEY (0x0004) – требуется для доступа к созданию подраздела ключа реестра.

KEY_ENUMERATE_SUB_KEYS (0x0008) – требуется для перечисления подразделов ключа реестра.

KEY_EXECUTE (0x20019) – то же, что и KEY_READ.

KEY_NOTIFY (0x0010) – требуется для запроса на сообщение об изменении для ключа реестра или подразделов ключа реестра.

KEY_QUERY_VALUE (0x0001) – требуется для доступа к получению значений ключа реестра.

KEY_READ (0x20019) – комбинация значений STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, и KEY_NOTIFY.

KEY_SET_VALUE (0x0002) – предназначен для доступа к реестру в режиме создания удаления или установки значения.

KEY_WOW64_32KEY (0x0200) – указывает на то, что приложение на 64-х разрядной ОС Windows должно оперировать с 32-х битным представлением реестра. Этот флаг должен быть скомбинирован, используя оператор OR, с другими флагами, которые отвечают за доступ к реестру. Windows 2000 не поддерживает этот флаг.

KEY_WOW64_64KEY (0x0100) – указывает на то, что приложение на 64-х разрядной ОС Windows должно оперировать с 64-х битным представлением реестра. Этот флаг должен быть скомбинирован, используя оператор OR, с другими флагами, или отправляют запрос, или обращаются к значениям реестра. Windows 2000 не поддерживает этот флаг.

KEY_WRITE (0x20006) – комбинация прав доступа STANDARD_RIGHTS_WRITE, KEY_SET_VALUE и KEY_CREATE_SUB_KEY.

phkResult – исходящий параметр, указатель на переменную, которая принимает описатель открытого ключа. Если ключ не один – из предопределенных ключей

реестра, после того как вы закончите с ним работать, необходимо вызвать функцию **RegCloseKey** для закрытия описателя открытого ключа.

Если функция выполнилась успешно, она возвращает значение **ERROR_SUCCESS**. В отличие от функции **RegCreateKeyEx** функция **RegOpenKeyEx** не создает специальных ключей, если ключа нет в реестре.

RegCloseKey – функция предназначена для закрытия описателя указанного ключа реестра.

Синтаксис функции:

```
LONG WINAPI RegCloseKey(
  __in  HKEY hKey
);
```

hKey – входящий параметр, описатель открытого ключа, который нужно закрыть;
этот описатель можно получить с помощью функций RegCreateKeyEx, RegCreateKeyTransacted, RegOpenKeyEx, RegOpenKeyTransacted, или RegConnectRegistry. Если функция выполнилась корректно, она возвращает значение ERROR_SUCCESS.

RegCreateKeyEx – функция создания указанного ключа реестра; если ключ уже существует, функция открывает его.

Синтаксис функции:

```
LONG WINAPI RegCreateKeyEx(
  __in       HKEY hKey,
  __in       LPCTSTR lpSubKey,
  __reserved  DWORD Reserved,
  __in_opt    LPTSTR lpClass,
```

```
  __in       DWORD dwOptions,
  __in       REGSAM samDesired,
  __in_opt   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  __out      PHKEY phkResult,
  __out_opt  LPDWORD lpdwDisposition
);
```

hKey – входящий параметр, описатель открытого ключа реестра, этот параметр может быть получен функцией RegCreateKeyEx или RegOpenKeyEx или может быть одним из следующих предопределенных разделов:

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;
HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpSubKey – входящий параметр, имя подраздела, который эта функция открывает или создает. Если этот параметр является указателем на пустую строку – phkResult получит новый описатель на ключ, определенный в параметре hKey. Этот параметр не может быть NULL.

Reserved – этот параметр зарезервирован и должен быть 0.

lpClass – входящий параметр, определенный пользователем тип этого раздела, параметр не обязательный и может быть опущен.

dwOptions – входящий параметр, может принимать одно из следующих значений:

REG_OPTION_BACKUP_RESTORE – если этот флаг

установлен, функция игнорирует параметр *samDesired* и пытается открыть ключ с правами достаточными для резервного копирования или восстановления раздела.

REG_OPTION_CREATE_LINK – указывает на то, что создаваемый параметр – символическая ссылка. Путь должен быть абсолютным путем реестра.

REG_OPTION_NON_VOLATILE – это значение, используемое по умолчанию, информация хранится в энергонезависимой памяти – файле и сохраняется при перезагрузке системы.

REG_OPTION_VOLATILE – все ключи, создаваемые функцией с этим параметром, будут храниться в памяти и не будут сохраняться при перезагрузке системы.

samDesired – входящий параметр, маска, определяющая, какие права доступа нужны для создания раздела.

lpSecurityAttributes – входящий параметр, указатель на структуру SECURITY_ATTRIBUTES, которая определяет, будет ли полученный описатель наследоваться дочерним процессом, если этот параметр NULL – описатель не наследуется.

phkResult – исходящий параметр, указатель на переменную, которая принимает описатель открытого или созданного ключа. Если ключ не один – из предопределенных ключей реестра, после того, как он нам больше не потребуется, необходимо вызвать функцию RegCloseKey для закрытия описателя открытого ключа.

lpdwDisposition – исходящий параметр, указатель на переменную, которая получает одно из 2-х значений: REG_CREATED_NEW_KEY – означает, что ключ не существовал ранее и был создан, и REG_OPENED_

EXISTING_KEY – означает, что ключ уже существовал и был открыт. Этот параметр не обязателен и может быть NULL.

Функция возвращает ERROR_SUCCESS в случае своего успешного завершения.

RegDeleteKey – функция удаляет подраздел и все содержащиеся в нем значения.

Синтаксис функции:

```
LONG WINAPI RegDeleteKey(
  __in  HKEY hKey,
  __in  LPCTSTR lpSubKey
);
```

hKey – входящий параметр, описатель открытого ключа реестра, этот параметр может быть получен функцией RegCreateKeyEx или RegOpenKeyEx или может быть одним из имен предопределенных разделов:

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;
HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpSubKey – входящий параметр, который нужно удалить. Этот параметр не может быть NULL.

Функция возвращает ERROR_SUCCESS в случае своего успешного завершения.

RegDeleteKeyEx – функция удаляет подраздел и все содержащиеся в нем значения.

Синтаксис функции:

```
LONG WINAPI RegDeleteKeyEx(
  __in       HKEY hKey,
```

```
  __in       LPCTSTR lpSubKey,
  __in       REGSAM samDesired,
  __reserved  DWORD Reserved
);
```

hKey – входящий параметр, описатель открытого ключа реестра, этот параметр может быть получен функцией RegCreateKeyEx или RegOpenKeyEx или может быть одним из следующих предопределенных разделов:

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;
HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpSubKey – входящий параметр, имя раздела, который нужно удалить. Этот параметр не может быть NULL.

samDesired – маска доступа, которая определяет специфическое для конкретной платформы представление реестра, может принимать одно из 2-х значений:

KEY_WOW64_32KEY – удаляет ключ из 32-х битного представления реестра;

KEY_WOW64_64KEY – удаляет ключ из 64-х битного представления реестра.

Reserved – параметр зарезервирован и должен быть 0.

Функция возвращает ERROR_SUCCESS в случае своего успешного завершения.

RegSetValueEx – функция устанавливает данные и тип определенного значения в ключе реестра.

Синтаксис функции:

```
LONG WINAPI RegSetValueEx(
  __in       HKEY hKey,
  __in_opt   LPCTSTR lpValueName,
  __reserved DWORD Reserved,
  __in       DWORD dwType,
  __in_opt   const BYTE *lpData,
  __in       DWORD cbData
);
```

hKey – входящий параметр, описатель открытого ключа реестра, должен быть открыт с правами KEY_SET_VALUE, этот параметр может быть получен функцией RegCreateKeyEx или RegOpenKeyEx или может быть одним из имен предопределенных разделов:

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;
HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpValueName *– входящий необязательный параметр, имя значения, которое нужно установить, если такого значения не существует в разделе, функция добавляет его в раздел. Если этот параметр NULL или пустая строка – функция устанавливает тип и данные для неименованного ключа или значения по умолчанию.*

Reserved – параметр зарезервирован и должен быть 0.

dwType – входящий параметр. Тип данных, указанный параметром lpData.

lpData *– входящий необязательный параметр, содержит в себе хранимые данные.*

cbData – размер информации (в байтах), на которую указывает параметр IpData.

Функция возвращает ERROR_SUCCESS в случае своего успешного завершения.

RegSetKeyValue – функция устанавливает данные для определенного значения в определенном ключе реестра и подразделе.

Синтаксис функции:

```
LONG WINAPI RegSetKeyValue(
  __in     HKEY hKey,
  __in_opt  LPCTSTR lpSubKey,
  __in_opt  LPCTSTR lpValueName,
  __in     DWORD dwType,
  __in_opt  LPCVOID lpData,
  __in     DWORD cbData
 );
```

hKey – входящий параметр, описатель открытого ключа реестра, должен быть открыт с правами KEY_SET_VALUE; этот параметр может быть получен функцией RegCreateKeyEx или RegOpenKeyEx или может быть одним из имен корневых разделов :

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;
HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpSubKey *– входящий необязательный параметр,*

имя ключа, должен быть подветкой ключа, определенного в параметре hKey.

lpValueName *– входящий необязательный параметр, имя значения реестра, данные которого нужно обновить.*

Reserved – параметр зарезервирован и должен быть 0.

dwType – входящий параметр, тип данных, указанный параметром lpData.

lpData *– входящий необязательный параметр, содержит в себе данные, которые будут храниться с указанным именем значения.*

cbData – размер информации (в байтах), указанный параметром lpData. Если данные REG_SZ, REG_EXPAND_SZ или REG_MULTI_SZ, cbData должен содержать размер завершающего нуль-символа или символов.

Функция возвращает ERROR_SUCCESS в случае своего успешного завершения

RegDeleteValue – функция удаляет именованное значение из указанного ключа реестра.

Синтаксис функции:

```
LONG WINAPI RegDeleteValue(
  __in     HKEY hKey,
  __in_opt  LPCTSTR lpValueName
);
```

hKey – входящий параметр, описатель открытого ключа реестра, должен быть открыт с правами доступа

KEY_SET_VALUE; этот параметр может быть получен функцией RegCreateKeyEx или RegOpenKeyEx или может быть одним из имен таких предопределенных разделов:

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;
HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpValueName *– входящий необязательный параметр, значение реестра, которое нужно удалить; если этот параметр NULL или пустая строка – удаляется значение, установленное функцией* ***RegSetValue.***

Функция возвращает ERROR_SUCCESS в случае своего успешного завершения.

RegDeleteKeyValue – функция удаляет определенное значение из определенного ключа реестра и подраздела.

Синтаксис функции:

```
LONG WINAPI RegDeleteKeyValue(
  __in     HKEY hKey,
  __in_opt  LPCTSTR lpSubKey,
  __in_opt  LPCTSTR lpValueName
);
```

hKey – входящий параметр, описатель открытого ключа реестра, должен быть открыт с правами KEY_SET_VALUE, этот параметр может быть получен функцией RegCreateKeyEx или RegOpenKeyEx или может быть одним из таких предопределенных разделов:

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;

HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpSubKey *– входящий необязательный параметр, определяет имя ключа, этот ключ должен быть подразделом ключа, определенного в параметре hKey.*

lpValueName *– входящий необязательный параметр, значение реестра, которое нужно удалить из ключа.*

Функция возвращает ERROR_SUCCESS в случае своего успешного завершения.

RegGetValue – функция получает тип и данные определенного значения реестра.

Синтаксис функции:

```
LONG WINAPI RegGetValue(
  __in        HKEY hkey,
  __in_opt    LPCTSTR lpSubKey,
  __in_opt    LPCTSTR lpValue,
  __in_opt    DWORD dwFlags,
  __out_opt   LPDWORD pdwType,
  __out_opt   PVOID pvData,
  __inout_opt LPDWORD pcbData
);
```

hKey – входящий параметр, описатель открытого ключа реестра, должен быть открыт с правами KEY_QUERY_VALUE, этот параметр может быть получен функцией RegCreateKeyEx или RegOpenKeyEx или может быть одним из таких предопределенных разделов:

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;
HKEY_LOCAL_MACHINE;
HKEY_USERS.

lpSubKey *– входящий необязательный параметр, имя ключа реестра, этот ключ должен быть подразделом ключа, указанного параметром hKey.*

dwFlags *– входящий необязательный параметр, флаги, которые указывают значение типа данных, которое мы хотим получить. Если значение типа данных не отвечает критерию этого параметра, функция завершается неудачей. Этот параметр может принимать следующие значения:*

RRF_RT_ANY – никаких ограничений на тип данных;

RRF_RT_DWORD – ограничивает функцию приемом данных только 32-bit REG_BINARY или REG_DWORD;

RRF_RT_QWORD – ограничивает функцию приемом данных только 64-bit REG_BINARY или REG_DWORD;

RRF_RT_REG_BINARY – ограничивает функцию приемом данных только REG_BINARY;

RRF_RT_REG_DWORD – ограничивает функцию приемом данных только REG_DWORD;

RRF_RT_REG_EXPAND_SZ – ограничивает функцию приемом данных только REG_EXPAND_SZ;

RRF_RT_REG_MULTI_SZ – ограничивает функцию приемом данных только REG_MULTI_SZ;

RRF_RT_REG_NONE – ограничивает функцию приемом данных только REG_NONE;

RRF_RT_REG_QWORD – ограничивает функцию приемом данных только REG_QWORD;

RRF_RT_REG_SZ – ограничивает функцию приемом данных только REG_SZ;

Этот параметр так же может содержать значения:

RRF_NOEXPAND – не расширять автоматически строки, если тип данных REG_EXPAND_SZ;

RRF_ZEROONFAILURE – если параметр pvData – не NULL, устанавливает содержимое буфера в ноль при неудачном завершении функции.

pdwType *– исходящий необязательный параметр, указатель на переменную, которая получает код, указывающий на тип хранимых данных в указанном значении. Параметр может быть NULL, если тип не требуется.*

pvData *– исходящий необязательный параметр, указатель на буфер, который получает значение данных.*

pcbData *– входящий, исходящий необязательный параметр, указатель на переменную, которая определяет размер буфера в байтах, указанный параметром pvData. Когда функция возвращается, эта переменная будет содержать размер данных, скопированных в pvData.*

Функция возвращает ERROR_SUCCESS в случае своего успешного завершения.

Теперь давайте рассмотрим пример программы, работающей с реестром.

Программа позволяет добавить ключ, добавить значение, удалить ключ, удалить значение реестра.

Рассмотрим участки кода, которые отвечают за эти действия:

```
//Переменная, хранящая открытый ключ реестра
HKEY phkResult = NULL;
//Функция открытия ключа реестра
//и если его нет, то ключ создается
int CreateKey(HKEY hRoot,char sKey[])
{


       //Попытка открыть ключ реестра
       LONG lResult = RegOpenKeyEx(hRoot,sKey,0,KEY _
                              ALL _ ACCESS,&phkResult);
       if(lResult != ERROR _ SUCCESS)
       {

       if (lResult == ERROR _ FILE _ NOT _ FOUND)

       {
```

```
        if(MessageBox(NULL,"Key Not Found. Create
                          it?",NULL,MB _ YESNO) == IDYES)

        {

        //Попытка создания ключа реестра

        lResult = RegCreateKeyEx(hRoot,sKey,0,NULL,REG _
OPTION _ NON _ VOLATILE,KEY _ ALL _ ACCESS,NULL,&ph-
kResult,NULL);

        if(lResult != ERROR _ SUCCESS)

        MessageBox(NULL,"Error Create Key",NULL,MB _ OK);

        else
        {


        MessageBox(NULL,"Key Created",NULL,MB _ OK);


        }

        }

        }

        else {

        MessageBox(NULL,"Error Open",NULL,MB _ OK);


        }



        }

        else
        {

        MessageBox(NULL,"Key Opened",NULL,MB _ OK);
        }
        return lResult;
}
```

Дальше в функции обработчике сообщений диалогового окна пишем:

```
//Переменные, которые нам потребуются для работы с
//реестром
      HWND hCombobox = GetDlgItem(hDlg,IDC_COMBO1);
      HWND hKey = GetDlgItem(hDlg,IDC_EDIT1);
      HWND hValName = GetDlgItem(hDlg,IDC_EDIT2);
      HWND hVal = GetDlgItem(hDlg,IDC_EDIT3);
      HKEY hRoot[4] = {HKEY_CLASSES_ROOT,
                       HKEY_CURRENT_USER,
                       HKEY_LOCAL_MACHINE,
                       HKEY_USERS};


      //Временные буферы для хранения данных
      char sRoot[255] = {'0'};
      char sKey[255] = {'0'};
      char sValName[255] = {'0'};
      char sVal[255] = {'0'};

      int iPos = (int)SendMessage(hCombobox,
                      CB_GETCURSEL,0,0);
      SendMessage(hCombobox,CB_GETLBTEXT,(
                  WPARAM)iPos,(LPARAM)(LPTSTR)sRoot);
      GetWindowText(hKey,sKey,2550);
      GetWindowText(hValName,sValName,2550);
      GetWindowText(hVal,sVal,2550);

      char sPath[1024] = {'0'};
      strncpy(sPath,sRoot,strlen(sRoot));
      strncat(sPath,"\\",1);
      strncat(sPath,sKey,strlen(sKey));


      switch (message)
      {
      case WM_INITDIALOG:

      //ComboBox Initialise
      SendMessage(hCombobox,CB_ADDSTRING,0,(LPARAM)
                (LPCTSTR)_ _T("HKEY_CLASSES_ROOT"));
```

```
SendMessage(hCombobox,CB_ADDSTRING,0,
          (LPARAM)__T("HKEY_CURRENT_USER"));

SendMessage(hCombobox,CB_ADDSTRING,0,(
          LPARAM)__T("HKEY_LOCAL_MACHINE"));

SendMessage(hCombobox,CB_ADDSTRING,0,
                  (LPARAM)__T("HKEY_USERS"));

SendMessage(hCombobox,CB_SETCURSEL,(WPARAM)0,0);

SetWindowText(hKey,TEXT("SOFTWARE\\TestDir"));


return (INT_PTR)TRUE;

case WM_COMMAND:

if (LOWORD(wParam) == IDOK || LOWORD(wParam) ==
                                        IDCANCEL)

{

//Если был открыт ключ реестра, закрываем его

if(phkResult != NULL)

RegCloseKey(phkResult);

EndDialog(hDlg, LOWORD(wParam));
return (INT_PTR)TRUE;

}


//Обработка нажатия кнопки Create Key

if(LOWORD(wParam) == IDC_BTN_CREATE_KEY &&
                   HIWORD(wParam) == BN_CLICKED)
```

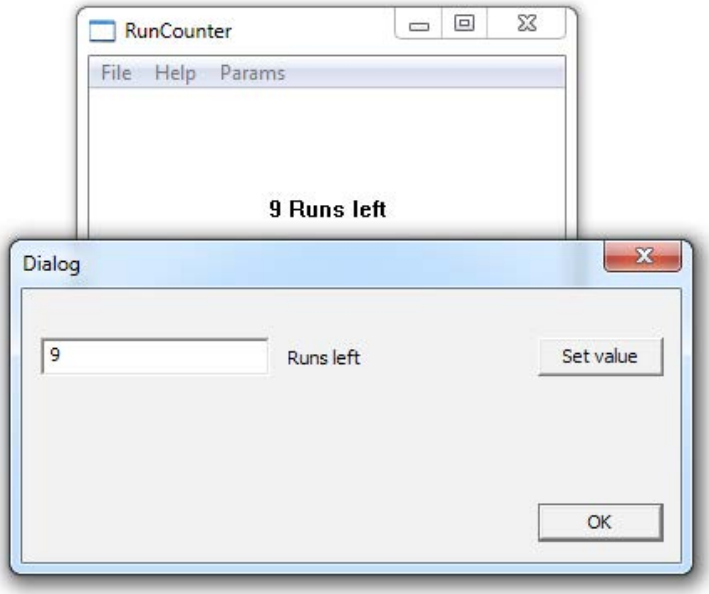
```
        {


        //Если был открыт ключ реестра, закрываем его

        if(phkResult != NULL)

        RegCloseKey(phkResult);

        CreateKey(hRoot[iPos],sKey);

        }


        //Обработка нажатия кнопки Create Val

        if(LOWORD(wParam) == IDC_BTN_CREATE_VAL &&
                          HIWORD(wParam) == BN_CLICKED)

        {

        //Если был открыт ключ реестра, закрываем его

        if(phkResult != NULL)

        RegCloseKey(phkResult);

        int lResult = CreateKey(hRoot[iPos],sKey);

        if(lResult == ERROR_SUCCESS)

        {

//Устанавливаем новое значение реестра

lResult = RegSetValueEx(phkResult,sValName,0,REG_SZ,
(const BYTE*)sVal,sizeof(sVal));

//Если функция отработала корректно, выводим
//соответствующее сообщение
```

```
if(lResult == ERROR _ SUCCESS)

{

MessageBox(NULL,"Create Value Success","",MB _ OK);

}

else

MessageBox(NULL,"Create Value Fail","",MB _ OK);

}

}


//Обработка нажатия кнопки Delete Key

if(LOWORD(wParam) == IDC _ BTN _ DEL _ KEY &&
              HIWORD(wParam) == BN _ CLICKED)

{

//Удаляем ветку реестра

int lResult = RegDeleteKey(hRoot[iPos],sKey);

//Если функция отработала корректно, выводим
//соответствующее сообщение

if(lResult == ERROR _ SUCCESS)

{

MessageBox(NULL,"Delete Key Success","",MB _ OK);

}

else
```

```
MessageBox(NULL,"Delete Key Fail","",MB _ OK);

}

//Обработка нажатия кнопки Delete Value

if(LOWORD(wParam) == IDC _ BTN _ DEL _ VAL &&
                  HIWORD(wParam) == BN _ CLICKED)

{


//Если был открыт ключ реестра, закрываем его

if(phkResult != NULL)

RegCloseKey(phkResult);


int lResult = CreateKey(hRoot[iPos],sKey);

      if(lResult == ERROR _ SUCCESS)

{


//Удаляем значение реестра

lResult = RegDeleteValue(phkResult,sValName);

if(lResult == ERROR _ SUCCESS)

{

MessageBox(NULL,"Delete Value Success","",MB _ OK);

}

else
```

```
        MessageBox(NULL,"Delete Value Fail","",MB _ OK);
        }
        }

        break;
        }
```

Рассмотрим еще один пример программы. Программа будет подсчитывать количество собственных запусков, и когда параметр реестра достигнет 0, программа перестанет запускаться. Так же добавим в программу возможность самостоятельно установить параметр в реестре, отвечающий за количество запусков.



```
/ RunCounter.cpp : Defines the entry point for the
application.
//

#include "stdafx.h"
#include "RunCounter.h"

#define MAX _ LOADSTRING 100

// Global Variables:
HINSTANCE hInst;
      // current instance
 TCHAR szTitle[MAX _ LOADSTRING];
      // The title bar text
TCHAR szWindowClass[MAX _ LOADSTRING];
      // the main window class name
HKEY phkResult = NULL;
HWND hVal;

// Forward declarations of functions included in this
// code module:
ATOM                    MyRegisterClass(HINSTANCE
hInstance);
BOOL                    InitInstance(HINSTANCE, int);
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
INT _ PTR CALLBACK About(HWND, UINT, WPARAM, LPARAM);
INT _ PTR CALLBACK SetParams(HWND, UINT, WPARAM,
LPARAM);
int CheckVal();
int SetVal(int);

int APIENTRY _ tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
      UNREFERENCED _ PARAMETER(hPrevInstance);
      UNREFERENCED _ PARAMETER(lpCmdLine);
```

```
        // TODO: Place code here.
        MSG msg;
        HACCEL hAccelTable;

        // Initialize global strings
        LoadString(hInstance, IDS _ APP _ TITLE, szTitle,
                                        MAX _ LOADSTRING);
        LoadString(hInstance, IDC _ RUNCOUNTER,
                        szWindowClass, MAX _ LOADSTRING);
        MyRegisterClass(hInstance);

        // Perform application initialization:
        if (!InitInstance (hInstance, nCmdShow))
        {

        return FALSE;
        }

        hAccelTable = LoadAccelerators(hInstance,
                      MAKEINTRESOURCE(IDC _ RUNCOUNTER));


        // Main message loop:
        while (GetMessage(&msg, NULL, 0, 0))
        {

        if (!TranslateAccelerator(msg.hwnd, hAccelTable,
                                                    &msg))

        {

        TranslateMessage(&msg);

        DispatchMessage(&msg);

        }
        }

        return (int) msg.wParam;
}
```

```
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//
//    This function and its usage are only necessary
//    if you want this code
//    to be compatible with Win32 systems prior to the
//    'RegisterClassEx'
//    function that was added to Windows 95. It is
//    important to call this function
//    so that the application will get 'well formed'
//    small icons associated
//    with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
      WNDCLASSEX wcex;

      wcex.cbSize = sizeof(WNDCLASSEX);

      wcex.style               = CS_HREDRAW | CS_
                                              VREDRAW;
      wcex.lpfnWndProc   = WndProc;
      wcex.cbClsExtra          = 0;
      wcex.cbWndExtra          = 0;
      wcex.hInstance           = hInstance;
      wcex.hIcon               = LoadIcon(hInstance,
      MAKEINTRESOURCE(IDI_RUNCOUNTER));
      wcex.hCursor       = LoadCursor(NULL, IDC_
                                               ARROW);
      wcex.hbrBackground= (HBRUSH)(COLOR_WINDOW+1);
      wcex.lpszMenuName  = MAKEINTRESOURCE(IDC_
                                            RUNCOUNTER);
      wcex.lpszClassName= szWindowClass;
      wcex.hIconSm       = LoadIcon(wcex.hInstance,
      MAKEINTRESOURCE(IDI_SMALL));
```

```
      return RegisterClassEx(&wcex);
}

//
//   FUNCTION: InitInstance(HINSTANCE, int)
//
//   PURPOSE: Saves instance handle and creates main
//   window
//
//   COMMENTS:
//
//        In this function, we save the instance
//        handle in a global variable and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Store instance handle in our
                      // global variable

   hWnd = CreateWindow(szWindowClass, szTitle,
   WS _ OVERLAPPEDWINDOW,
      500, 300, 300, 200, NULL, NULL, hInstance, NULL);
   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//
//  FUNCTION: WndProc(HWND, UINT, WPARAM, LPARAM)
```

```
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND   - process the application menu
//  WM_PAINT     - Paint the main window
//  WM_DESTROY   - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
      int wmId, wmEvent;
      PAINTSTRUCT ps;
      HDC hdc;
      int val = 0;
      char buf[255];
      RECT rect;
      char* text = " Runs left";
      switch (message)
      {
      case WM_COMMAND:

      wmId    = LOWORD(wParam);

      wmEvent = HIWORD(wParam);

      // Parse the menu selections:

      switch (wmId)

      {

      case IDM_ABOUT:

      DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX),
      hWnd, About);

      break;

      case IDM_EXIT:
```

```
        DestroyWindow(hWnd);

        break;

        case ID _ PARAMS _ SETPARAMS:


        DialogBox(hInst, MAKEINTRESOURCE(IDD _ DIALOG1),
        hWnd, SetParams);

        break;

        default:

        return DefWindowProc(hWnd, message, wParam,
               lParam);

        }

        break;
        case WM _ PAINT:

        hdc = BeginPaint(hWnd, &ps);



        GetClientRect(hWnd, &rect);

        val = CheckVal();


        _ itoa(val,buf,10);

        strncat _ s(buf, _ countof(buf),text, _
        countof(buf)- strlen(buf));


        DrawText(hdc, buf, -1, &rect,

        DT _ SINGLELINE | DT _ CENTER | DT _ VCENTER );
```

```
        EndPaint(hWnd, &ps);

        break;
        case WM_CREATE:

        //Проверяем сколько осталось попыток запуска
        //программы

        val = CheckVal();

        if(val > 0)

        //Устанавливаем на 1 попытку меньше

        SetVal(--val);

        //Если попыток не осталось
        else

        {

        //Выводим сообщение

        MessageBox(NULL,"Value is 0",NULL,MB_OK);


        //и не даем программе запуститься -
        //уничтожаем окно программы

        DestroyWindow(hWnd);

        }

        break;
        case WM_DESTROY:

        PostQuitMessage(0);

        break;
        default:
```

```
        return DefWindowProc(hWnd, message, wParam,
                                lParam);
        }
        return 0;
}

//Функция проверки значения реестра
int CheckVal()
{
        DWORD pcbData = sizeof(DWORD);
        DWORD data = 10;

        LONG lResult = RegOpenKeyEx(HKEY _
LOCAL _ MACHINE,"SOFTWARE\\TestDir",0,KEY _ ALL _
ACCESS,&phkResult);
        if(lResult != ERROR _ SUCCESS)
        {

        if (lResult == ERROR _ FILE _ NOT _ FOUND)

        {

        if(MessageBox(NULL,"Key Not Found. Create
                        it?",NULL,MB _ YESNO) == IDYES)

        {

        //Попытка создания ключа реестра

        lResult = RegCreateKeyEx(HKEY _ LOCAL _
MACHINE,"SOFTWARE\\TestDir",0,NULL,REG _ OPTION _ NON _
VOLATILE,KEY _ ALL _ ACCESS,NULL,&phkResult,NULL);

        if(lResult != ERROR _ SUCCESS)

        {

        MessageBox(NULL,"Error Create Key",NULL,MB _ OK);

        return NULL;
```

```
        }

        else

        {

        MessageBox(NULL,"Key Created",NULL,MB _ OK);


        lResult =
        RegSetValueEx(phkResult,"TestRuns",0,REG _
        DWORD,(BYTE*)&data,sizeof(DWORD));

        }

        }

        }

        else

        {

        MessageBox(NULL,"Error Open",NULL,MB _ OK);


        return NULL;

        }

        }

        else
        {

        lResult = RegGetValue(HKEY _ LOCAL _
MACHINE,"SOFTWARE\\TestDir","TestRuns",RRF _ RT _ REG _
DWORD,NULL,&data,&pcbData);

        if (lResult != ERROR _ SUCCESS)
```

```
	{

	lResult =
	RegSetValueEx(phkResult,"TestRuns",0,REG _
	DWORD,(BYTE*)&data,sizeof(DWORD));

	}

	}
	return data;

}

//Функция установки значения реестра
int SetVal(int val)
{
	DWORD pcbData = sizeof(DWORD);
	DWORD data = val;

	//Открываем ветку реестра
	LONG lResult = RegOpenKeyEx(HKEY _
LOCAL _ MACHINE,"SOFTWARE\\TestDir",0,KEY _ ALL _
ACCESS,&phkResult);
	if(lResult != ERROR _ SUCCESS)
	{

	MessageBox(NULL,"Error Open",NULL,MB _ OK);


	return NULL;


	}

	else
	{


	//Получаем значение из параметра реестра
```

```
        lResult =
        RegSetValueEx(phkResult,"TestRuns",0,REG _
        DWORD,(BYTE*)&data,sizeof(DWORD));

        if (lResult != ERROR _ SUCCESS)

        {

        MessageBox(NULL,"Error Set Value",NULL,MB _ OK);

        }
        }
        return data;
}

INT _ PTR CALLBACK SetParams(HWND hDlg, UINT message,
WPARAM wParam, LPARAM lParam)
{
        UNREFERENCED _ PARAMETER(lParam);
        hVal = GetDlgItem(hDlg,IDC _ EDIT1);
        char Val[255] = {'0'};
        char* buf = new char[255];
        int val = CheckVal();
        switch (message)
        {
        case WM _ INITDIALOG:

        buf = _ itoa(val,buf,10);

        SetWindowText(hVal,buf);

        return (INT _ PTR)TRUE;

        case WM _ COMMAND:

        if (LOWORD(wParam) == IDOK || LOWORD(wParam) ==
                                      IDCANCEL)

        {
```

```
        EndDialog(hDlg, LOWORD(wParam));

        return (INT_PTR)TRUE;

        }

        //Обработчик нажатия кнопки Set Val

        if (LOWORD(wParam) == IDC_BT_SETVAL &&
            HIWORD(wParam) == BN_CLICKED)

        {

        //Получаем значение из окна

        GetWindowText(hVal,Val,255);

        val = atoi(Val);

        //и устанавливаем его в параметр реестра

        if(SetVal(val) != NULL)

        MessageBox(NULL,"Succes",NULL,MB_OK);

        }
        break;
        }
        return (INT_PTR)FALSE;
}

// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message,
WPARAM wParam, LPARAM lParam)
{
        UNREFERENCED_PARAMETER(lParam);
        switch (message)
        {
        case WM_INITDIALOG:
```

```
        return (INT_PTR)TRUE;

        case WM_COMMAND:

        if (LOWORD(wParam) == IDOK || LOWORD(wParam) ==
                                      IDCANCEL)

        {

        EndDialog(hDlg, LOWORD(wParam));

        return (INT_PTR)TRUE;

        }

        break;
        }
        return (INT_PTR)FALSE;
}
```

5. Работа с реестром с помощью платформы Microsoft .Net

Работа с системным реестром с помощью платформы **Microsoft .Net** реализуется в пространстве имен **Microsoft.Win32**. В нем предусмотрен набор типов, при помощи которых считать данные из реестра или внести в него какие-либо изменения, можно очень легко и быстро. Перечень типов для работы с системным реестром представлен в таблице.

Тип Microsoft.Win32	Назначение
`Класс Registry`	Высокоуровневая абстракция всего реестра, которая обеспечивает объекты RegistryKey, предоставляющие корневые разделы в реестре Windows, и методы static для доступа к парам "раздел-значение".
`Класс RegistryKey`	Тип, при помощи которого производится вставка новых параметров (ключей) в реестр, их удаление и изменение.
`перечисление RegistryHive`	Перечисление, в котором хранятся названия каждой ветви реестра.

Тип Microsoft.Win32	Назначение
перечисление RegistryKeyPermissionCheck	Определяет, выполняются ли проверки безопасности при открытии разделов реестра и доступе к соответствующим парам "имя-значение".
перечисление RegistryValueKind	Определяет типы данных, используемые для хранения значений в реестре, или задает тип данных значения в реестре.
перечисление RegistryValueOptions	Определяет необязательное поведение при возвращении пар "имя-значение" из раздела реестра.

Рассмотрим класс **Registry**, который предоставляет набор стандартных корневых разделов, находящихся в реестре компьютеров, работающих под управлением **Windows**. Реестр является средством хранения сведений о приложениях, пользователях и стандартных системных параметрах. Например, приложения используют реестр для хранения сведений, которые необходимо сохранить после закрытия приложения и к которым необходимо получать доступ при перезагрузке приложения. Например, можно сохранять цветовые настройки, положение или размер окна. Для разных пользователей эти сведения могут сохраняться в различных местах реестра.

Экземпляры **RegistryKey**, показываемые классом **Registry**, создают схему базового механизма хранения в реестре вложенных разделов и значений. Все разделы доступны только для чтения, так как реестр зависит от

их существования. Класс **Registry** предоставляет доступ к разделам, приведенным в таблице.

Тип	Назначение
`CurrentUser`	Сохраняет сведения о пользовательских параметрах.
`LocalMachine`	Сохраняет сведения о конфигурации для локального компьютера.
`ClassesRoot`	Сохраняет сведения о типах (и классах) и их свойствах.
`Users`	Сохраняет сведения о стандартной пользовательской конфигурации.
`PerformanceData`	Сохраняет сведения о производительности для программных компонентов.
`CurrentConfig`	Сохраняет сведения об оборудовании, не являющемся специфиче.ским для пользователя
`DynData`	Сохраняет динамические данные.

Определив корневой раздел реестра, в котором необходимо сохранять и из которого необходимо извлекать сведения, можно использовать класс **RegistryKey**, чтобы добавлять или удалять вложенные разделы и выполнять действия со значениями данного раздела.

Аппаратные устройства с помощью интерфейса PnP автоматически помещают сведения в реестр. Драйвера устройств записывают сведения в реестр посредством стандартных интерфейсов API.

Класс **Registry** также содержит два статических метода для установки **SetValue** и получения **GetValue** значений разделов реестра. При каждом своем использовании эти

методы открывают и закрывают разделы реестра, поэтому при работе с большим количеством значений их производительность ниже, чем у аналогичных методов класса **RegistryKey**.

Рассмотрим класс **RegistryKey**, который представляет узел уровня раздела в реестре **Windows**. Этот класс является инкапсуляцией реестра. Для получения его экземпляра необходимо использовать один из статических методом класса **Registry** (см. предыдущую таблицу).

Некоторые члены класса **RegistryKey** представлены в таблице:

Имя	Назначение
`Close`	Если содержимое раздела было изменено, следует закрыть раздел и записать его на диск.
`CreateSubKey`	Создает новый вложенный раздел или открывает существующий вложенный раздел.
`DeleteSubKey`	Удаляет заданный вложенный раздел.
`DeleteSubKeyTree`	Рекурсивно удаляет вложенный раздел и все дочерние вложенные разделы.
`DeleteValue`	Удаляет заданное значение из этого раздела.
`Flush`	Записывает в реестр все атрибуты заданного открытого раздела реестра.
`GetAccessControl`	Возвращает безопасность элемента управления доступом для текущего раздела реестра.
`GetSubKeyNames`	Возвращает массив строк, который содержит все имена вложенных разделов.

Имя	Назначение
GetValue	Возвращает значение, связанное с заданным именем.
GetValueKind	Возвращает тип данных реестра для значения, связанного с заданным именем.
GetValueNames	Возвращает массив строк, содержащий все имена значений, связанных с этим разделом.
OpenRemoteBaseKey	Открытие нового раздела RegistryKey, который представляет запрошенный раздел на удаленном компьютере.
OpenSubKey	Возвращает заданный вложенный раздел.
SetAccessControl	Применяет безопасность управления доступом Windows к существующему разделу реестра.
SetValue	Устанавливает значение пары "имя-значение" в разделе реестра.
Свойство Name	Возвращает имя раздела.
Свойство SubKeyCount	Возвращает количество вложенных разделов для текущего раздела.
Свойство ValueCount	Возвращает число значений в разделе.

Рассмотрим пример получения вложенных разделов для выбранного раздела системного реестра и вывода их имен на консоль.

```
using System;
using Microsoft.Win32;

class ShowRegistry
{
    public static void Main()
    {
        int SelectItem;
        RegistryKey [] regs = new []
                        {Registry.ClassesRoot,
                         Registry.CurrentUser,
                         Registry.LocalMachine,
                         Registry.Users,
                          Registry.CurrentConfig};
        do {
            int i = 1;
            Console.WriteLine("Выберите раздел
                               системного реестра");
            foreach (RegistryKey reg in regs)
                Console.WriteLine("{0}. {1}",i++,reg.
                                                Name);
            Console.WriteLine("0. Выход");
            Console.Write("> ");
            SelectItem = Convert.ToInt32
                         (Console.ReadLine()[0]) - 48;
            Console.WriteLine();
            if (SelectItem > 0 && SelectItem <= regs.
                                            GetLength(0))
                PrintRegKeys(regs[SelectItem - 1]);
        } while (SelectItem != 0);
    }
    static void PrintRegKeys(RegistryKey rk)
    {
        string[] names = rk.GetSubKeyNames();
        Console.WriteLine("Подразделы {0}:",rk.Name);
        Console.Write
               Line("--------------------------------");
        foreach (string s in names)
            Console.WriteLine(s);
        Console.Write
               Line("--------------------------------");
    }
}
```

Следующий пример создает раздел реестра, в котором сохраняются значения нескольких типов данных, а затем выполняется возвращение и отображение значений.

```
using System;
using Microsoft.Win32;

public class GetSetValue
{
    public static void Main()
    {
        string user = Registry.CurrentUser.Name,
               skey = "GetSetValue",
               skeyName = user + "\\" + skey;
        // Запись
        Registry.SetValue(skeyName,"",0x1234);
        // по умолчанию
        Registry.SetValue(skeyName,"GSVQWord",
                          0x0123456789ABCDEF,
                           RegistryValueKind.QWord);
        Registry.SetValue(skeyName,"GSVString",
                                    "Путь: %path%");
        Registry.SetValue(skeyName,"GSVExpand
                          String","Путь: %path%",
                           RegistryValueKind.
                                              ExpandString);
        Registry.SetValue(skeyName,"GSVArray",
                           new[] {"Лента 1",
                                  "Лента 2","Лента 3"});
        // Считывание
        Console.WriteLine("GSVNotExists: {0}",
            (string)Registry.GetValue(skeyName,
                                      "GSVNotExists",
                                       "GSVNotExists
                                        отсутствует!"));
```

```
        RegistryKey rk = Registry.CurrentUser.OpenSub
                                                Key(skey);
        string[] rks = rk.GetValueNames();
        foreach (string s in rks)
        {
            if (s.Length == 0) Console.Write
                                        ("(Default): ");
            else Console.Write("{0}: ", s);
            if (rk.GetValueKind(s) ==
        RegistryValueKind.MultiString)
            {
                foreach (string subs in (string[])
                                        rk.GetValue(s))
                    Console.Write("\t\"{0}\"",subs);
                Console.WriteLine("\n");
            }
            else Console.WriteLine(rk.GetValue(s));
        }
        Console.WriteLine("Посмотрите в реестре
                          введенные значения " +
                          "и нажмите Enter");
        Console.ReadLine();
        // Удаление
        Registry.CurrentUser.DeleteSubKey(skey);
    }
}
```

6. Разработка динамически подключаемых библиотек с использованием .Net Framework

Все приложения, созданные до этого момента, представляли собой программы, состоящие из одного файла **EXE**. Приложения в целом могут состоять из нескольких исполняемых файлов, что связано с возможностью повторного использования кода.

Проблема повторного использования кода стоит уже очень давно. До появления платформы **Microsoft .Net**, наиболее популярным способом решения этой проблемы являлось применение **COM**-серверов. Разработчику приходилось затрачивать много сил, чтобы создать для **COM**-сервера всю необходимую инфраструктуру.

В **.Net** все намного проще, поскольку для двоичных файлов используется новый формат, называемый сборкой (**assembly**).

Приложения **.Net** создаются путем объединения любого количества сборок – двоичных файлов (**DLL** или **EXE**), содержащих в себе номер версии, метаданные (информацию о себе), а также типы и дополнительные ресурсы.

Одномодульная модель, применяемая ранее отличается простотой, но обладает и рядом недостатков:

- исполняемый модуль может разрастаться до больших размеров;
- при каждом обновлении потребуется повторная сборка всей программы;
- каждый исполняемый модуль будет иметь свои экземпляры функций, версии которых могут различаться;
- для достижения наилучшей производительности в различных средах может потребоваться использование различных версий программы, в которых применяются различные методики.

Библиотеки **DLL** используются практически в любой операционной системе. Они обеспечивают возможность решения этих проблем:

- библиотечные функции не связываются во время компоновки – связывание осуществляется во время загрузки программы (неявное связывание) или во время ее выполнения (явное связывание);
- DLL могут использоваться для создания общих библиотек (shared libraries) – одну и ту же библиотеку DLL могут совместно использовать несколько одновременно выполняющихся программ, но в память будет загружена только одна ее копия;
- новые версии программ или другие возможные варианты их реализации могут поддерживаться путем простого предоставления новой версии DLL;
- в случае явного связывания решение о том, какую версию библиотеки использовать, программа может принимать во время выполнения.

.Net DLL обязательно должна экспортировать только одну функцию – **DllMain()**. Основное отличие сборок **.Net** от двоичных файлов серверов **COM** заключается в том, что сборки содержат не платформенно-зависимые инструкции, а код на промежуточном языке (**Microsoft Intermediate Language**, **MSIL** или просто **IL**). Этот язык не зависит ни от платформы, ни от типа центрального процессора и компилируется в платформенно-зависимые инструкции только во время выполнения.

Каждая сборка **.Net** содержит в себе информацию о каждом типе сборки и каждом члене каждого типа. Эта информация генерируется полностью автоматически. Кроме того, любая сборка **.Net** содержит манифест – набор метаданных о самой сборке. Манифест содержит информацию о всех двоичных файлах, которые входят в состав данной сборки, номере версии сборки, а также сведения обо всех внешних сборках, на которые ссылается данная сборка.

Любая сборка может состоять из нескольких модулей. Модуль – это файл сборки **.Net**. Сборки иногда называют "логическими DLL". Чаще всего сборка состоит из единственного файла. В этом случае между "логическим двоичным файлом" – сборкой и "физическим двоичным файлом" – модулем сборки существует отношение "один к одному". Вся информация сборки будет расположена в единственном физическом файле.

Сборка из одного файла

Основной целью создания сборки из нескольких файлов является более эффективная загрузка приложения. Если, например, удаленный клиент обращается к многофайловой сборке, состоящей из трех файлов по одному мегабайту и ему нужны возможности, реализуемые в одном файле, то время загрузки этого файла будет меньше, чем всей сборки.

Многофайловые сборки объединяются с помощью манифеста, в котором хранится информация обо всех модулях данной сборки. Манифест может быть расположен в отдельном файле, но гораздо чаще он помещается в один из модулей сборки.

Типы и ресурсы, находящиеся внутри сборки, могут совместно использоваться самыми разными приложениями. Но в **.Net** можно не только разрешить повторное использование кода, но и явно запретить его. Это достигается путем разделения сборок на открытые и частные (см. далее в этом разделе).

В **.Net** типы идентифицируются, в том числе и по сборке, в которой они находятся. Таким образом, если в

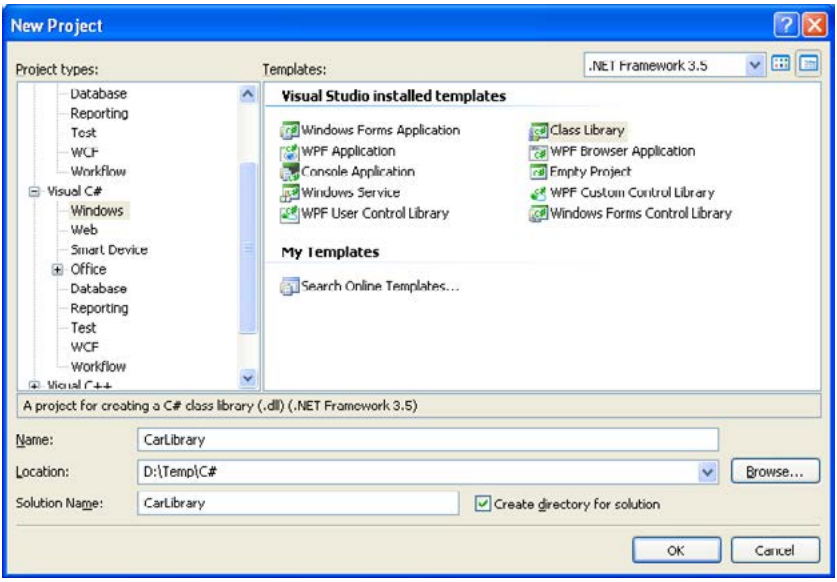
приложении используются две сборки, в каждой из которых есть тип с одинаковыми названиями, конфликта имен не возникнет.

У каждой сборки есть свой идентификатор версии (**version identifier**), применяющийся ко всем типам и ресурсам внутри каждого модуля сборки. Используя эту информацию, среда выполнения **.Net** гарантирует загрузку нужной версии сборки. Идентификатор состоит из двух частей: текстовой строки (**informational version**) и цифрового идентификатора (**compatibility version**).

Цифровой идентификатор состоит из четырех цифр, разделенных точками:

- основной номер версии (major version);
- дополнительный номер версии (minor version);
- номер сборки (build number);
- номер редакции (revision number).

Создадим динамически подключаемую библиотеку кода с использованием **.Net Framework** на C#. Например, она будет содержать абстрактный класс **Car** – базовый для создания иерархий машин в какой-то игре. Физически эта библиотека кода будет представлять собой однофайловую сборку с именем **CarLibrary**. Чтобы приступить к созданию библиотеки, необходимо выбрать в интегрированной среде разработки **Visual Studio .Net** новый проект **Class Library**.



Создадим абстрактный класс **Car**, в котором будет определен набор внутренних членов, определенных как **protected**:

```
namespace CarLibrary
{
    using System;
    public abstract class Car
    {
        protected Int16 curSpeed;
        protected Int16 maxSpeed;
        public Car()
        {
        }
        public Car(Int16 max, Int16 cur)
        {
            maxSpeed = max;
            CurSpeed = cur;
        }
```

```
        public Int16 CurSpeed
        {
            get { return curSpeed; }
            set
            {
                if (value <= maxSpeed) curSpeed =
                                              value;
                else curSpeed = maxSpeed;
            }
        }
        public Int16 MaxSpeed
        {
            get { return maxSpeed; }
        }
        public abstract void Up();
        public abstract void Down();
    }
}
```

Создадим два класса, непосредственно производных от класса **Car**: **Bus** и **Truck**. Каждый из них реализует абстрактные методы **Up()** и **Down()**:

```
public class Bus : Car
{
    Int16 passengers;
    Int16 maxPassengers;
    public Bus() { }
    public Bus(Int16 max, Int16 cur, Int16 maxpass,
                                     Int16 pass)
        : base(max, cur)
    {
        maxPassengers = maxpass;
        Passengers = pass;
    }
    public Int16 Passengers
    {
```

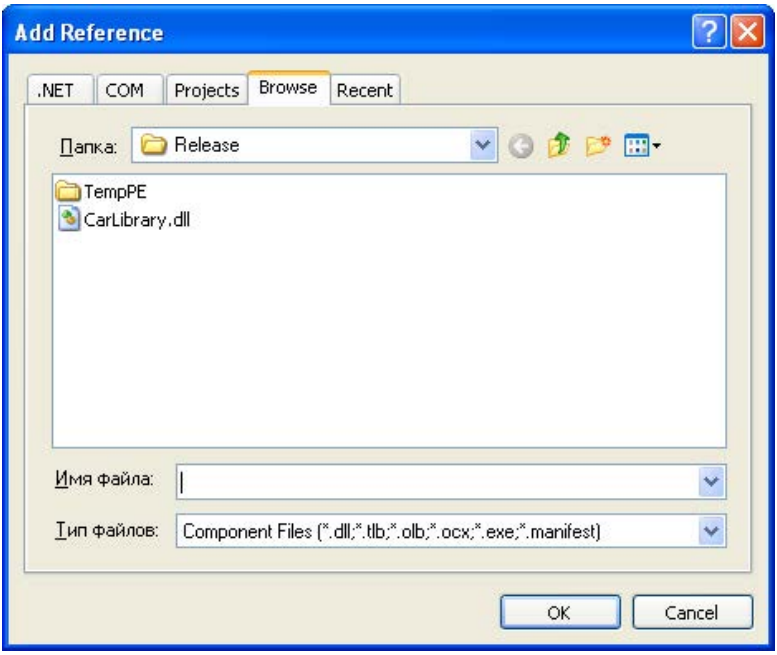
```
        get { return passengers; }
        set
        {
            if (value <= maxPassengers) passengers =
                                                value;
            else passengers = maxPassengers;
        }
    }
    public Int16 MaxPassengers
    {
        get { return maxPassengers; }
    }
    public override void Up()
    {
        Passengers++;
    }
    public override void Down()
    {
        Passengers--;
    }
}
public class Truck : Car
{
    Int16 cargo;
    Int16 maxCargo;
    public Truck() { }
    public Truck(Int16 max, Int16 cur, Int16 maxcrg,
                                       Int16 crg)
        : base(max, cur)
    {
        maxCargo = maxcrg;
        Cargo = crg;
    }
    public Int16 Cargo
    {
        get { return cargo; }
        set
        {
```

```
            if (value <= maxCargo) cargo = value;
            else cargo = maxCargo;
        }
    }
    public Int16 MaxCargo
    {
        get { return maxCargo; }
    }
    public override void Up()
    {
        Cargo = MaxCargo;
    }
    public override void Down()
    {
        Cargo = 0;
    }
}
```

Откомпилируем созданную библиотеку кода.

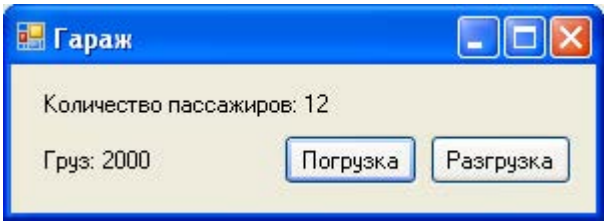
Поскольку и класс **Bus**, и класс **Truck** объявлены как **public**, их можно использовать в любых приложениях. Создадим клиентское приложение C#, использующее эти классы.

Первое, что нужно сделать – с помощью диалогового окна **Add References** включить в список ссылок для приложения только что скомпилированную библиотеку **CarLibrary.dll**, выбрав ее при помощи закладки **Browse**.



Добавив в проект ссылку на сборку **CarLibrary.dll**, можно гарантировать, что при первой же попытке запустить приложение интегрированная среда разработки **Visual Studio .NET** создаст полную копию **CarLibrary.dll** в каталоге **Debug**.

После создания ссылки на сборку **CarLibrary.dll** можно использовать содержащиеся в ней типы. В клиентском приложении создадим объект класса автобус (Bus). Создадим механизм рандомной посадки и высадки пассажиров (независимо от действий пользователя). Также создадим объект класса грузовик (Truck). Предоставим пользовательский интерфейс для добавления груза в грузовик или для извлечения груза из грузовика. На рисунке показан скриншот приложения.



Текст клиентского приложения на C# будет выглядеть следующим образом:

```
using System;
using System.Windows.Forms;
using CarLibrary;
using System.Threading;
namespace Cars
{
    public partial class Form1 : Form
    {
        Bus bus = new Bus(70, 60, 25, 14);
        Truck truck = new Truck(90, 40, 2000, 500);
        System.Windows.Forms.Timer tim;
        Thread [] tbus;
        Random delay = new Random();
        public Form1()
        {
            InitializeComponent();
            lTruck.Text += truck.Cargo;
            tbus = new [] {new Thread(
                           new ThreadStart(UpBus)),
                           new Thread
                            (new ThreadStart(DownBus))};
            tbus[0].Start();
            tbus[1].Start();
            tim = new System.Windows.Forms.Timer();
            tim.Interval = 10;
            tim.Tick += new EventHandler(tim _ Tick);
            tim.Start();
```

```
        }
        void UpBus()
        {
            while (true)
            {
                lock (this)
                {
                    bus.Up();
                }
                Thread.Sleep(delay.Next(500) + delay.
                                                Next(500));
            }
        }
        void DownBus()
        {
            while (true)
            {
                lock (this)
                {
                    bus.Down();
                }
                Thread.Sleep(delay.Next(500) + delay.
                                                Next(500));
            }
        }
        void tim _ Tick(object sender, EventArgs e)
        {
            lBus.Text = "Количество пассажиров: " +
                         bus.Passengers;
        }
        private void bUp _ Click(object sender,
                                 EventArgs e)
        {
            truck.Up();
            lTruck.Text = "Груз: " + truck.Cargo;
        }
        private void bDown _ Click(object sender,
                                   EventArgs e)
        {
```

```
            truck.Down();
            lTruck.Text = "Груз: " + truck.Cargo;
        }
        private void Form1 _ FormClosed(object sender,
FormClosedEventArgs e)
        {
            tbus[0].Abort();
            tbus[1].Abort();
        }
    }
}
```

Это приложение очень похоже на все примеры, рассмотренные ранее. Единственное отличие заключается в том, что в нем используется специально заготовленная внешняя библиотека типов.

Любая сборка **.Net** может быть либо частной (**private**), либо сборкой для общего доступа (**shared**) – третьего не дано. У обеих версией сборок есть как общие черты, так и различия. Общих черт гораздо больше: у всех сборок **.Net** одинаковая структура и схожее содержание (например, оба типа сборок предоставляют доступ к своим открытым членам). Различия между этими типами сборок состоят в особенностях их именования, политики версий и размещения сборок на компьютере пользователя.

Частные сборки **.Net** – это наборы типов, которые могут быть использованы только теми приложениями, в состав которых они входят. Например, библиотека **CarLibrary.dll** – это частная библиотека.

Как и частные сборки, сборки для общего доступа – это набор типов и необязательных ресурсов внутри модулей – двоичных файлов сборки. Главное различие

между типами сборок заключается в том, что частные сборки предназначены для использования одним приложением (в состав которого они и входят), а сборки для общего доступа – для использования неограниченным количеством приложений на клиентском компьютере.

Как правило, сборки для общего доступа устанавливаются не в каталог приложения, а в специальный каталог, называемый глобальным кэшем сборок (**Global Assembly Cache**, **GAC**). Этот каталог расположен в каталоге **<имя_диска>:\Windows\Assembly**.

Еще одно различие между различными типами сборок заключается в том, что в сборках для общего доступа используется дополнительная информация о версии. Эта дополнительная информация называется "общим именем" или "сильным именем". Кроме того, для сборок этого типа среда выполнения не игнорирует информацию о версии, а активно использует ее.

При создании сборки для общего доступа обязательно потребуется создать уникальное общее имя. Само это имя состоит из следующей информации:

- дружественное текстовое имя и "культурная информация" (информация о естественном языке);
- идентификатор версии;
- пара открытый/закрытый ключ;
- цифровая подпись.

Создание общего имени основывается на криптографии открытого ключа. При создании сборки для общего доступа необходимо создать пару открытый/закрытый ключ. Эта пара будет использована **.Net**-со-

вместимым компилятором, который затем поместит маркер открытого ключа в манифест сборки (пометив его тегом **[.publickeytoken]**). Закрытый ключ не помещается в манифест сборки. Вместо этого он используется для создания цифровой подписи, которая помещается в сборку. Сам закрытый ключ будет храниться в том модуле сборки, в котором находится манифест.

Создание пары открытый/закрытый ключ для сборки производится при помощи утилиты **sn.exe** (от **strong name** – "сильное имя"). У этой утилиты есть множество параметров командной строки, среди них **–k**, который используется для создания набора ключей и сохранения их в указанном нами файле с расширением ***.snk**.

Чтобы сборка стала общей, необходимо снабдить ее парой открытый/закрытый ключ из созданного файла ***.snk**. Это делается очень просто: необходимо в качестве значения атрибута **AssemblyKeyFile** (в файле **AssemblyInfo.cs**) указать полный путь к созданному файлу ***.snk**:

```
[assembly: AssediblyKeyfile(@"D:\Shared\key.snk")]
```

Все остальное сделает компилятор при создании сборки. Открытый ключ, который помещается в манифест сборки для общего пользования, можно просмотреть при помощи **ILDasm.exe**.

После того как сборка для общего пользования готова, последнее, что осталось сделать – поместить ее в глобальный кэш сборок. Проще всего сделать это, просто перетащив мышью файл сборки в каталог **GAC**. Другой вариант – воспользоваться утилитой **gacutil.exe**. Для того чтобы устанавливать сборки в **GAC** или удалять их из него, необходимо обладать правами администратора.

После того как сборка для общего пользования установлена в **GAC**, она готова к использованию любыми приложениями. Удаление сборки из **GAC** производится предельно просто: нужно удалить нужный файл из каталога.

7. Создание хуков в приложениях платформы Microsoft .Net

Давайте вспомним некоторую информацию из курса WinAPI. Каким образом операционная система Windows управляет всеми приложениями, системными и пользовательскими, которые выполняются на работающем компьютере? Это происходит благодаря сообщениям. Сообщение Windows – это объект структуры такого вида:

```
typedef struct tagMSG {
  HWND   hwnd;
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
} MSG, *PMSG, *LPMSG;
```

Первое поле этой структуры hwnd, это дескриптор окна, которому предназначено сообщение. Давайте запомним, что каждое сообщение в Windows имеет своего адресата. Любое системное действие и любое действие пользователя приводят к возникновению одного или нескольких сообщений. После возникновения сообщение чаще всего попадает в специальную fifo очередь, называемую очередью сообщений (*message queue)*. Оче-

редь сообщений – это системный объект, который создается и обслуживается операционной системой (ОС). ОС просматривает сообщения, собранные в очереди сообщений и отправляет каждое сообщение тому адресату, который указан в его поле hwnd.

Есть еще другой тип сообщений, которые не попадают в очередь сообщений, а направляются прямо тому окну, чей дескриптор указан в hwnd.

Адресатом сообщения является окно, которое принадлежит какому-нибудь приложению. У каждого Windows приложения есть своя собственная очередь сообщений, куда попадают сообщения, отправленные из очереди сообщений. А у окна есть оконная процедура, умеющая обрабатывать полученные сообщения. Именно в оконной процедуре создается реакция приложения на обрабатываемое сообщение. Итак, запомните, что все действия приложений в операционной системе Windows инициализируются сообщениями.

Поэтому управление сообщениями является в Windows чрезвычайно важным. Давайте предположим, что мы смогли каким-то образом получить сообщение до того, как оно попало своему адресату, т.е. окну какого-то приложения. Мы знаем, что в полях wParam и lParam находятся данные сообщения. Перехватив само сообщение, мы получим эти данные, сможем их прочитать или изменить, изменив таким образом поведение приложения, которое получит это сообщение. Более того, мы можем вообще уничтожить перехваченное сообщение, чем тоже изменим поведение приложения. Так

можно ли перехватывать сообщения до того, как они попали адресату? Да, можно. Для этого используются хуки.

Windows делит все сообщения на более чем десять типов. Например, сообщения, возникающие при работе мышки или при работе клавиатуры, сообщения, адресованные пунктам меню и т.п. Во внутренней реализации Windows для каждого типа сообщений встроены так называемые hook-точки. Каждая такая точка содержит список указателей (hook chain) на специальные callback функции, называемые фильтрующими функциями. Каждая фильтрующая функция имеет такой вид:

```
LRESULT CALLBACK HookProc(int nCode,WPARAM
wParam,LPARAM lParam)
{
        // check whether to process the message
        if (code < 0)
                return CallNextHookEx(hookDeleg,
                                code, wParam, lParam);
        // Process the message
        some code here
        // Pass the message to the next filtering
        // function
                return CallNextHookEx(hookDeleg,
                                code, wParam, lParam);
}
```

Параметр nCode – это код hook-точки, который подключаемая процедура использует для идентификации действия. Функция хука использует его, чтобы определить действие, которое необходимо исполнить. Значение кода хука зависит от типа хука; каждый тип имеет

свой набор кодов хука. Значения параметров wParam и lParam зависят от кода хука, но обычно они содержат информацию об отправленном сообщении.

Обратите внимание на параметр nCode. Если его значение меньше 1, то текущая фильтрующая процедура не должна обрабатывать сообщение, а должна передать его следующей функции в цепочке вызовов. В противном случае, сообщение обрабатывается и затем опять передается на обработку следующей функции в цепочке вызовов. Имя фильтрующей функции может быть произвольным, спецификатор HookProc приведен в качестве примера.

Если в hook-точке какого-то типа размещена такая цепочка вызовов фильтрующих функций и при этом в системе возникает сообщение этого типа, то Windows сначала передаст это сообщение на обработку каждой из фильтрующих функций в цепочке вызовов. А уже потом это сообщение (если ни одна из фильтрующих функций его не удалит) попадет своему адресату. Возможно, в измененном виде. Действие, которое может совершить фильтрующая функция, зависит от типа данного хука. Функции для некоторых типов хуков могут только просматривать сообщения; другие функции могут изменять сообщения или останавливать их продвижение по цепочке, не позволяя им достигнуть следующей функции хука или окна-адресата.

Если hook-точка может перехватывать все сообщения своего типа, независимо от того, из какого процесса они отправлены, то такая hook-точка называется глобальной. Еще есть hook-точки, которые могут пере-

хватывать сообщения только одного какого-то потока. Однако запомните, что .NET Framework не поддерживает глобальные хуки, кроме WH_KEYBOARD_LL и WH_MOUSE_LL.

Чтобы добавить фильтрующую функцию в hook-точку применяется функция SetWindowsHookEx(). Эта функция устанавливает фильтрующую процедуру в начало цепочки вызова.

Хуки несколько замедляют работу системы, поскольку они увеличивают количество действий, которые производит система при обработке каждого сообщения. В связи с этим следует устанавливать хуки только при необходимости и удалять их, как только они становятся не нужными.

В следующей таблице показана область видимости некоторых хуков и их назначение. За более детальной информацией обращайтесь к специальной литературе.

Хук	Область видимости	Назначение
LWNDPROC	Поток или вся система	Вызывается при каждом вызове функции SendMessage. Функции-фильтру передается код хука, показывающий, была ли произведена посылка сообщения из текущего потока, а также указатель на структуру с информацией о сообщении.
WH_CBT	Поток или вся система	Необходим при разработке приложений для интерактивного обучения. Получает код хука, показывающий, какое произошло событие, и соответствующие этому событию данные.

Хук	Область видимости	Назначение
WH_DEBUG	Поток или вся система	Вызывается перед вызовом какой-либо фильтрующей функции. Она не может изменять значения, переданные этому хуку, но может предотвратить вызов фильтрующей функции, возвратив ненулевое значение.
WH_GETMESSAGE	Поток или вся система	Вызывается перед выходом из функций GetMessageи PeekMessage. Фильтрующие функции получают указатель на структуру с сообщением, которое затем (вместе со всеми изменениями) посылается приложению, вызвавшему GetMessage или PeekMessage.
WH_JOURNALRECORD	Только система	Вызывается при удалении события из системной очереди. Таким образом, фильтры этого хука вызываются для всех мышиных и клавиатурных событий, кроме тех, которые проигрываются регистрационным хуком на воспроизведение. Фильтрующие функции могут обработать сообщение (то есть, записать или сохранить событие в памяти, на диске, или и там, и там), но не могут изменять или отменять его.
WH_JOURNALPLAYBACK	Только система	Используется для посылки клавиатурных и мышиных сообщений таким образом, как будто они проходят через системную очередь. Основное назначение этого хука – проигрывание событий, записанных с помощью хука WH_JOURNALRECORD, но его можно также с успехом использовать для посылки сообщений другим приложениям.

Хук	Область видимости	Назначение
WH_FOREGROUNDIDLE	Поток или вся система	Вызывается, когда к текущему потоку не поступает пользовательский ввод для обработки. Когда хук установлен как локальный, Windows вызывает его только при условии отсутствия пользовательского ввода у потока, к которому прикреплен хук. Данный хук является уведомительным.
WH_SHELL	Поток или вся система	Вызывается при определенных действиях с окнами верхнего уровня (с окнами, не имеющими владельца). Когда хук установлен как локальный, он вызывается только для окон, принадлежащих потоку, установившему хук. Этот хук является информирующим, поэтому фильтры не могут изменять или отменять событие.
WH_KEYBOARD	Поток или вся система	Вызывается, когда функции GetMessage или PeekMessage собираются вернуть сообщения WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, WM_SYSKEYDOWN, или WM_CHAR. Когда хук установлен как локальный, эти сообщения должны поступать из входной очереди потока, к которому прикреплен хук. Фильтрующая функция получает виртуальный код клавишии состояние клавиатуры на момент вызова клавиатурного хука. Фильтры имеют возможность отменить сообщение.

Хук	Область видимости	Назначение
WH_MOUSE	Поток или вся система	Вызывается после вызова функций GetMessage или PeekMessage при условии наличия сообщения от мыши. Подобно хуку WH_KEYBOARD фильтрующие функции получают код-индикатор удаления сообщения из очереди (HC_NOREMOVE), идентификатор сообщения мыши и координаты x и y курсора мыши. Фильтры имеют возможность отменить сообщение.
WH_MSGFILTER	Поток или вся система	Вызывается, когда диалоговое окно, информационное окно, полоса прокрутки или меню получают сообщение, либо когда пользователь нажимает комбинацию клавиш ALT+TAB (или ALT+ESC) при активном приложении, установившем этот хук.
WH_SYSMSGFILTER	Только система	Идентичен хуку WH_MSGFILTER за тем исключением, что он имеет системную область видимости.

Для любого данного типа хука, первыми вызываются хуки потоков, и только затем системные хуки.

Если к одному хуку прикреплено несколько функций-фильтров, операционная система реализует очередь функций, реализованную по принципу **FILO** (**first input – last output**).

Платформа **Microsoft .Net** не поддерживает работу с хуками. Для работы с хуками необходимо импортиро-

вать функции из **Win32 API**, которые находятся в библиотеке **User32.dll**.

Чтобы создать хук необходимо сначала создать конкретную фильтрующую функцию повторного вызова.

В C# для создания подключаемой процедуры используются делегаты:

```
private delegate IntPtr HookProc(int nCode,
                                 IntPtr wParam,
                                 IntPtr lParam);
```

Затем с помощью функции SetWindowsHookEx() установить созданную фильтрующую функцию в цепочку вызовов, связанную с данным хуком.

Функция **SetWindowsHookEx()** всегда устанавливает фильтрующую функцию в начало цепочки хуков. Когда происходит событие, которое отслеживается данным типом хука, **Windows** вызывает функцию, которая находится в начале цепочки, связанной с этим хуком. Каждая фильтрующая функция в цепочке решает, передавать ли событие следующей функции. Функция хука может передать событие следующей функции, используя функцию **CallNextHookEx()**.

Обратите внимание, что функции хука для некоторых типов хуков могут только просматривать сообщения. **Windows** передает сообщения к каждой функции хука такого типа даже в том случае, если функции хука не используют функцию **CallNextHookEx()**.

Еще раз повторим, что hook-точка может быть глобальной. В этом случае она просматривает сообщения для всех потоков в системе. Можно также определить

hook-точку для одного потока, чтобы просматривать сообщения, предназначенные только этому потоку. Фильтрующие функции глобального хука должны вызываться из контекста любого приложения, поэтому они должна быть помещены в отдельный модуль библиотеки **DLL**. Фильтрующие функции, определенные для одного потока, вызываются только в контексте данного потока. Если приложение устанавливает фильтрующую функцию для одного из своих собственных потоков, она может находиться или в том же модуле, что и остальная часть кода приложения, или также в **DLL**. Если приложение устанавливает фильтрующую функцию для потока другого приложения, ее надо обязательно поместить в **DLL**.

Примечание.

Использовать глобальные хуки следует только в целях отладки, обычно лучше избегать их. Глобальные хуки снижают эффективность системы и вызывают конфликты с другими приложениями, которые используют такой же тип глобального хука.

Функцию **SetWindowsHookEx()** и другие следует импортировать из **user32.dll**. Она имеет следующий синтаксис:

```
IntPtr SetWindowsHookEx(int idHook,
                        HookProc lpfn,
                        IntPtr hMod,
                        uint dwThreadId);
```

Параметры имеют такой смысл: **idHook** – тип **hook**-процедуры (**WH_MOUSE**, **WH_KEYBOARD**, …); **lpfn** – подключаемая процедура, **callback**-функция; **hMod** –

handle для **DLL**, в которой находится и обрабатывается хук (если нужно сделать локальный хук, этот параметр нужно поставить в **null**); **dwThreadId** – идентификатор потока, с которым ассоциировать подключаемую процедуру. Если нужно контролировать все потоки, этот параметр должен быть 0.

Для завершения работы с хуками необходимо использовать функцию:

```
bool UnhookWindowsHookEx(IntPtr hhk);
```

Разработаем программу, которая будет глобально блокировать клавишу **Windows**. Для этого воспользуемся хуком **WH_KEYBOARD**:

```
using System;
using System.Diagnostics;
using System.Windows.Forms;
using System.Runtime.InteropServices;
class BlockWindows
{
    private const int WH_KEYBOARD_LL = 13;
    private const int WM_KEYDOWN = 0x0100;
    private static HookProc proc = HookCallback;
    private static IntPtr hook = IntPtr.Zero;
    public static void Main()
    {
        hook = SetHook(proc);
        Application.Run();
        UnhookWindowsHookEx(hook);
    }
    private static IntPtr SetHook(HookProc proc)
    {
        using (Process curProcess = Process.
                                    GetCurrentProcess())
        using (ProcessModule curModule = curProcess.
                                     MainModule)
        {
```

```
            return SetWindowsHookEx(WH_KEYBOARD_LL,
                                    proc,
                GetModuleHandle(curModule.ModuleName),
                                    0);
        }
    }
    private delegate IntPtr HookProc(int nCode, IntPtr
                                     wParam,
                                     IntPtr lParam);
    private static IntPtr HookCallback(int nCode,
                                       IntPtr wParam,
                                       IntPtr lParam)
    {
        if ((nCode >= 0) && (wParam == (IntPtr)WM_
                                               KEYDOWN))
        {
            int vkCode = Marshal.ReadInt32(lParam);
            if (((Keys)vkCode == Keys.LWin) || ((Keys)
                                  vkCode == Keys.RWin))
            {
                Console.WriteLine("{0} blocked!", (Keys)
                                                  vkCode);
                return (IntPtr)1;
            }
        }
        return CallNextHookEx(hook, nCode, wParam,
                                                 lParam);
    }
    [DllImport("user32.dll", CharSet = CharSet.Auto,
                                  SetLastError = true)]
    private static extern IntPtr SetWindowsHookEx
                                       (int idHook,
                                        HookProc lpfn,
                                        IntPtr hMod,
                                        uint dwThreadId);
    [DllImport("user32.dll", CharSet = CharSet.Auto,
                                   SetLastError = true)]
```

```
    private static extern bool
                    UnhookWindowsHookEx(IntPtr hhk);
    [DllImport("user32.dll", CharSet = CharSet.Auto,
                                    SetLastError = true)]
    private static extern IntPtr CallNextHookEx
                                   (IntPtr hhk,
                                    int nCode,
                                    IntPtr wParam,
                                    IntPtr lParam);
    [DllImport("kernel32.dll", CharSet = CharSet.Auto,
                                    SetLastError = true)]
    private static extern IntPtr
                  GetModuleHandle(string lpModuleName);
}
```

Для работы с мышиным хуком достаточно в предыдущем примере поменять константы **WH_KEYBOARD_LL = 13** на **WH_MOUSE_LL = 14**, указать эту константу при установке хука и при обработке событий обрабатывать мышиные сообщения.

Например, необходимо ограничить движение мыши по координате **X** слева не меньше, чем на **400 px**. Создаем новую фильтрующую функцию:

```
private const int WM_MOUSEMOVE = 0x0200;
...
private static IntPtr HookCallback(int nCode, IntPtr
                                wParam, IntPtr lParam)
{
    if ((nCode >= 0) && (wParam == (IntPtr)
                                    WM_MOUSEMOVE))
    {
        int X = Marshal.ReadInt32(lParam);
        if (X < 400) return (IntPtr)1;
    }
    return CallNextHookEx(hook, nCode, wParam, lParam);
}
```

8. Домашнее задание

Создать приложение а-ля "Total Commander". Обязательными являются возможности многопоточного поиска, а также установки глобальных комбинаций клавиш (с помощью хуков), при нажатии на которые будет активизироваться та или иная функциональность приложения.

Урок № 3

Небезопасный код, управление памятью, использование реестра, создание dll модулей