МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение высшего образования «Казанский национальный исследовательский технический
университет им. А.Н. Туполева-КАИ»
(КНИТУ-КАИ)
Чистопольский филиал «Восток»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

по дисциплине

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Индекс по учебному плану: **Б1.В.ДВ.01.01**

Направление подготовки: **09.03.01 Информатика и вычислительная техника**

Квалификация: **Бакалавр**

Профиль подготовки: **Автоматизированные системы обработки информации и управления**

Типы задач профессиональной деятельности: **проектный, производственно-технологический**

Рекомендовано УМК ЧФ КНИТУ-КАИ

Чистополь
2023 г.

Лабораторная работа №1

Работа с делегатами и событиями

По своей структуре делегат - это объект, который ссылается на метод.

С помощью делегата можно вызвать метод, на который он указывает.

В процессе выполнения программы делегату можно присвоить ссылку на другой метод. Это дает возможность определять во время выполнения программы, какой из методов должен быть вызван.

Делегаты реализуются как экземпляры классов, производных от библиотечного класса System.Delegate. Для создания делегата необходимо выполнит два шага.

На первом шаге необходимо объявить делегат. При этом сигнатура делегата должна полностью соответствовать сигнатуре метода, который он представляет.

Например, делегат должен ссылаться на статический метод класса CA:

```
static  int min(int x,int y),
```

тогда объявление делегата может выглядеть, следующим образом:

```
delegate int  LpFunc(int a,int b);
```

На втором шаге мы должны создать экземпляр делегата для хранения сведения о представляемом им методе:

```
LpFunc pfnk = new LpFunk(CA.min);
```

Экземпляр делегата может ссылаться на любой статический метод или метод объекта любого класса, при условии, что сигнатура метода полностью соответствует сигнатуре делегата.

Пример 1:

```
 using System;
namespace ConsoleApplication14
{
        class MathOprt
        {
                public static double Mul2(double val)
                {
                        return val*2;
                }
                public static double Sqr(double val)
                {
                        return val*val;
                }
        }
        delegate double DblOp(double x);//объявление делегата
        class Class1
        {
    static void Main(string[] args)
                {
                        DblOp [] operation = // создание экземпляров делегата
                        {
                                new DblOp(MathOprt.Mul2),
                                new DblOp(MathOprt.Sqr)
                        };
                        for(int j=0;j<operation.Length;j++)
                        {
                                Console.WriteLine("Резльтаты
операции[{0}]:",j);
                                Prc(operation[j], 4.0);
```

```
                    Prc(operation[j], 9.94);
                    Prc(operation[j], 3.143);
                }
            }
            static void Prc(DblOp act, double val)
            {
            double rslt = act(val);
            Console.WriteLine("Исходное значение {0}, результат  {1}",
                    val,rslt);
            }
        }
    }
```

Делегаты могут хранить несколько адресов областей памяти. То есть делегат может указывать на несколько различных методов. Это позволяет, последовательно инициализируя адреса вызывать метод за методом. Эта способность делегатов называется *многоадресностью делегатов*.

Для создания цепочки вызовов методов необходимо сначала создать экземпляр делегата для одного метода, а затем с помощью операции “ + = ” добавить остальные методы. В процессе выполнения кода можно не только добавлять новые методы, но и удалять не нужные с помощью операции ”- =”.

Методы, представляемые многоадресными делегатами должны возвращать значение void.

Перепишем код из примера 1 с применением многоадресного делегата.

Пример 2:

```
using System;
namespace ConsoleApplication14
{
```

```
class MathOprt
{
public static void Mul2(double val)
 {
  double rslt= val*2;
  Console.WriteLine("Mul2 bсходное значение {0},результат  {1}",
                        val,rslt);
 }
public static void Sqr(double val)
 {
  double rslt = val*val;
  Console.WriteLine("Sqr исходное значение {0}, результат  {1}",
                        val,rslt);
 }
}
delegate void DblOp(double x);//объявление делегата

class Class1
{
        [STAThread]
        static void Main(string[] args)
        {
                DblOp  operations = new DblOp(MathOprt.Mul2);
                        operations += new DblOp(MathOprt.Sqr);
                        Prc(operations, 4.0);
                        Prc(operations, 9.94);
                        Prc(operations, 3.143);
        }
        static void Prc(DblOp act, double val)
```

```
            {
                Console.WriteLine("\n**********\n");
                act(val);
            }
        }
    }
```

События

Работа с событиями осуществляется в C# согласно модели «издатель-подписчик». Класс, ответственный за инициализацию (выработку) событий публикует событие, и любые классы могут подписаться на это событие. При возникновении события исполняющая среда уведомляет всех подписчиков о произошедшем событии, при этом вызываются соответствующие методы- обработчики событий подписчиков. Какой обработчик события будет вызван – определяется делегатом.

Платформа .NET требует для всех обработчиков событий следующей сигнатуры кода:

```
void OnRecChange(object source, ChangeEventArgs e)
{
    // Код для обработки события
}
```

Обработчики событий обязательно имеют тип возвращаемого значения void. Обработчики событий принимают два параметра. Первый параметр _ это ссылка на объект, сгенерировавший событие. Эта ссылка передается обработчику самим генератором событий. Второй параметр – это ссылка на объект класса EventArgs или класса производного от него. В производном классе может содержаться дополнительная информация о событии.

Обработчик события определяется делегатом. Согласно сигнатуре обработчика события делегат должен принимать два параметра и выглядеть следующим образом:

public delegate void ChangeEventHandle(object source,ChangeEventArgs e);

Для того, чтобы иметь возможность подписаться на событие класс генератора событий должен содержать член типа указанного делегата с ключевым словом event и метод, который будет вызываться при возникновении события, например:

public event ChangeEventHandler OnChangeHandler;

Этот член является специализированной формой многообъектного делегата. Использую операцию “+=”, клиенты могут подписаться на это сообщение:

```
gnEvent.OnChangeHandler +=
new GenEvent.ChangeEventHandler (OnRecChange);
```

где gnEvent имя класса генератора событий.

Нижеследующий пример демонстрирует работу с событиями.

Пример№3:

```
using System;
namespace sobit
{
        class ChangeEventArgs : EventArgs
```

```
{
        string str;
        public string Str
        {
                get
                {
                        return str;
                }
        }
        int change;
        public int Change
        {
                get
                {
                        return change;
                }
        }
        public ChangeEventArgs(string str,int change)
        {
                this.str = str;
                this.change = change;
        }
}
        class GenEvent      // Генератор событий - издатель
        {
                public delegate void ChangeEventHandler
                        (object source,ChangeEventArgs e);

                public event ChangeEventHandler OnChangeHandler;
```

```
                public void UpdateEvent(string str,int change)
                {
                    if(change==0)
                        return;
                    ChangeEventArgs e =
                    new ChangeEventArgs(str,change);

                    if (OnChangeHandler != null)
                        OnChangeHandler(this,e);
                }
            }
        //Подписчик
        class RecEvent
        {
        //Обработчик события
            void OnRecChange(object source,ChangeEventArgs e)
            {
            int change = e.Change;
            Console.WriteLine("Вес груза '{0}' был {1} на {2} тонны",
                    e.Str,change > 0 ? "увеличен" : "уменьшен",
                    Math.Abs(e.Change));
            }
        // в конструкторе класса осуществляется подписка
            public RecEvent(GenEvent gnEvent)
            {
            gnEvent.OnChangeHandler += //здесь осуществляется
```
подписка

```
                                    new
GenEvent.ChangeEventHandler(OnRecChange);
                }
        }
        class Class1
        {
                [STAThread]
                static void Main(string[] args)
                {

                        GenEvent gnEvent = new GenEvent();
                        RecEvent inventoryWatch =     new RecEvent(gnEvent);
                        gnEvent.UpdateEvent("грузовика", -2);
                        gnEvent.UpdateEvent("автопоезда", 4);
                }
        }
}
```

Вопросы:

Можно ли по сигнатуре объявления делегата определить сигнатуру функции, которую представляет делегат.

Какие ограничения накладываются на функции, которые может представлять многоадресный делегат?

Как включить или исключить заданную функцию из списка функций, представляемых многоадресным делегатом?

Как объявляется событие?

Что такое событие?

Чем отличается событие от многоадресного делегата?

Какова общепринятая сигнатура обработчика события?

Как осуществляется генерация события?

Каким образом обработчику события передается дополнительная информация о произошедшем событии?

Как осуществляется «подписка» на событие?

Задания:

Создать приложение, в котором генератор события “снабжает” событие следующей информацией: название поезда, время прибытия, номер вагона и места. Приемник события распечатывает эту информацию.

Создать приложение, в котором генератор события “снабжает” событие следующей информацией: название поезда, станция назначения, станция отправления и время в пути. Приемник события распечатывает эту информацию.

Создать приложение, в котором генератор события после генерации первого события генерирует последующие события только в том случае, если приемник события уведомляет, что сообщение принято (квитирование). Для квитирования использовать первый параметр обработчика события.

Создать приложение, в котором генератор события после генерации первого события генерирует только определенное количество событий. Количество генераций определяется путем уведомления со стороны приемника. Для уведомления использовать первый параметр обработчика события.

Создать приложение, в котором генератор события после генерации первого события генерирует последующие события только в том случае, если приемник события уведомляет, что событие принято (квитирование). Для квитирования использовать второй параметр обработчика события.

Создать приложение, в котором генератор события после генерации первого события генерирует только определенное количество событий. Количество генераций определяется путем уведомления со стороны

приемника. Для уведомления использовать второй параметр обработчика события.

Создать приложение, в котором генератор события может генерировать три разных события. Приемники событий выступают в качестве абонентов почтового отделения и могут пересылать друг другу информацию, используя генератор в качестве почтового ящика. При этом они указывают номер (от 1 до 3) следующего приемника и некоторое целое число, которое передается получателю. Такой цикл передачи продолжается до тех пор, пока какой либо из приемников в качестве получателя не укажет номер ноль. В этом случае приложение завершает свою работу. При запуске приложения первое почтовое извещение всегда получает от генератора первый приемник. Для адресации и передачи информации использовать второй аргумент обработчика события.

Выполнить задание по пункту 7, но адресации и передачи информации использовать первый аргумент обработчика события.

Создать приложение, в котором генератор события, путем генерации одного события запрашивает у трех приемников некоторый ресурс. Каждый приемник сообщает, какое количество ресурса он может выделить. Для передачи информации использовать второй аргумент обработчика события.

Выполнить задание по пункту 9, но для передачи информации использовать первый аргумент обработчика события.

Лабораторная работа №2

Интерфейсы. Работа с коллекциями

Множество однотипных значений можно хранить в массиве, но для этой же цели можно применять и другие объекты – коллекции. У коллекций не фиксирован объем, их можно пополнять неограниченно. Существует несколько типов коллекций, и мы можем выбрать именно тот тип, что быстрее выполнит нужные нам операции над данными.

Итак, коллекция – это объект, а тип коллекции – это класс. Классы коллекций расположены в пространстве System.Collections.

Начнем с коллекции, которая называется ArrayList. Поведением она очень напоминает массив объектов objet[], только безразмерный.

Внутри объекта этой коллекции сидит массив, который постепенно заполняется. Как только все места будут заняты, создастся новый массив вдвое большего размера и данные из старого будут скопированы в него. Подмена внутреннего массива происходит незаметно для программиста, и создается впечатление, что массив безразмерный.Как только он обращается к элементу коллекции, который еще не был в нее добавлен, появляется исключение.

Пример: Создадим объект коллекции, добавим в нее несколько значений и распечатаем содержимое:

```
using System.Collections;
…
  static void Main(string[] args) {
    ArrayList m = new ArrayList();
    m.Add(5);
    m.Add(10);
    m.Add("строка");
    for (int i = 0; i < m.Count; i++) {
      Console.WriteLine(m[i]);       }
```

```
        m[2] = 5;
        m[10] = 5;  // будет исключение и кружка воды
    }
```

Конструктор без параметров всегда создает пустую коллекцию, но можно сконструировать и не пустую, передав в конструктор массив или другую коллекцию.

```
ArrayList m = new ArrayList(new object[] {20, 30, "stroka" });
```

Новые элементы лучше всего добавлять методом Add() в конец коллекции, но можно и в середину – методом Insert().

```
m.Insert(1, 25);
```

В первую позицию вставлено число 25, все, кто стоял правее, потеснились на шаг вправо. Нумерация элементов коллекции ведется с нуля.

m.RemoveAt(3); - этим методом мы убрали из коллекции элемент, стоящий на третьем месте. Остальные элементы сомкнули ряд.

Свойство *Count* хранит текущее число элементов коллекции. Оно нам пригодилось, чтобы организовать цикл. Свойство *Count* – только для чтения.

Вы заметили, что в коллекцию добавлялись элементы разных типов. Как ни странно, сходство коллекции ArrayList с массивом объясняется вовсе не тем, что внутри нее сидит массив, а тем, что она реализует интерфейс IList.

Интерфейс – это множество методов, и даже не самих методов, а только их заголовков.

И если кто-то скажет, что согласно спецификации C#, в интерфейс могут входить еще свойства и индексаторы, то и свойство, и индексатор – это всего лишь пара методов, поэтому первая фраза не далека от истины.

Если взять абстрактный класс и сделать абстрактными все его методы (свойства, индексаторы), то как раз получится интерфейс, поэтому в C++

такие классы успешно играют роль интерфейсов. Почему в C# ввели новое слово *interface?*

Объявлен интерфейс IList.

```
namespace System.Collections
{ public interface IList : ICollection, IEnumerable {
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
    int Add(object value);
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);                } }
```

Заголовки методов коллекции ArrayList. Слова *класс ArrayList реализует интерфейс IList* означают, что:

1) в объявлении класса ArrayList присутствует имя IList, вот так:

```
public class ArrayList : IList,ICollection,IEnumerable,ICloneable {…
```

2) класс ArrayList ОБЯЗАН иметь все члены, перечисленные в объявлении интерфейса IList.

С этим обязательством - очень строго, если хоть самый маленький метод не будет реализован, компилятор это не пропустит!!!.

Интерфейсы могут наследовать друг друга, например, IList наследует сразу два: ICollection и IEnumerable. Наследование означает, что, кроме членов, явно перечисленных в объявлении IList, в него входят члены еще двух интерфейсов. В отличие от классов языка C# интерфейсы допускают множественное наследование Абстрактный класс может содержать код

реализации, а интерфейс – никогда, поэтому лучше провести между ними четкую границу.

Еще один важный момент, на который следует обратить внимание: объект класса, который реализует некоторый интерфейс, можно неявно привести к типу этого интерфейса. Например: IList m = new ArrayList();

После этого через *m* можно вызвать только методы данного интерфейса. Такое приведение особенно кстати, когда объект создается не прямым вызовом конструктора, а возвращается каким-то методом. IList m = GetList();

В данном случае истинный тип объекта скрыт от клиента.

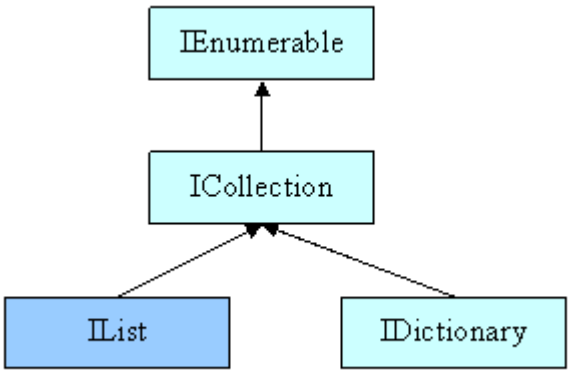
Реализация интерфейса – не совсем то же самое, что полиморфное наследование абстрактного класса. При полиморфном наследовании набор методов предка стараются не расширять, при реализации интерфейса такой задачи обычно не ставится.

Так коллекция ArrayList имеет полезные методы, которые не вошли в интерфейс IList. Например, метод *void AddRange(ICollection m),*
который добавляет не один элемент, а сразу массив или другую коллекцию.

Сходство ArrayList с массивом объектов объясняется наличием индексатора в интерфейсе IList: object this[int index] { get; set; }.

Вообще, любой класс, реализующий IList, будет похож на массив объектов. Поэтому говорят, что интерфейс IList представляет последовательные коллекции, такие, в которых каждый элемент имеет номер, от 0 до Count – 1.

Рассмотрим иерархию наследования интерфейсов из пространства



System.Collections

Интерфейс ICollection

Любой класс, реализующий этот интерфейс, уже является коллекцией.

```
public interface ICollection : IEnumerable {
    int Count { get; }
    void CopyTo(Array array, int index);
    bool IsSynchronized { get; }
    object SyncRoot { get; }    }
```

У интерфейса есть свойство *Count* – текущее число элементов коллекции, метод *CopyTo()*, выгружающий коллекцию в массив, и еще пара свойств, необходимых для многопоточной работы с коллекцией. ICollection наследует интерфейс IEnumerable.

Интерфейс IEnumerable

В нем только один метод

public interface IEnumerable {IEnumerator GetEnumerator(); }

он возвращает объект, при помощи которого можно последовательно посетить все элементы коллекции. Каждый, кто хочет пройтись по коллекции, вызывает ее метод *GetEnumerator()*, получает свой объект и гуляет с его помощью.

Но, чтобы перебирать элементы *ArrayList*, нам было достаточно было одной целой переменной. Вот так мы распечатывали все коллекцию.

```
IList list = new ArrayList(new object[] { "1", "2", "3", });
for (int i = 0; i < list.Count; ++i) {Console.WriteLine(list[i]); }
```

Переменная *i* хранила номер очередного элемента и позволяла обратиться к нему – *list[i]*.

Дело в том, что коллекции бывают разные,только те, что реализуют *IList*, позволяют клиенту такие вольности, ведь в *IList* есть индексатор. Универсальное решение, пригодное для всех видов коллекций, это – получить у коллекции объект-проводник и с его помощью переходить от одного элемента к другому.

Интерфейс IEnumerator

В нем есть свойство *Current* – текущий элемент коллекции, метод *MoveNext()*, вызов которого заставляет свойство *Current* ссылаться уже на следующий элемент, и метод *Reset(),* который устанавливает *Current* на самый первый элемент коллекции. Вот его определение.

```
public interface IEnumerator {object Current { get; }
        bool MoveNext();void Reset();  }
```

Для сравнения пройдемся с его помощью по коллекции *ArrayList*.

```
for (IEnumerator e = list.GetEnumerator(); e.MoveNext(); ) {
    Console.WriteLine(e.Current);     }
```

Немного громоздко, зато универсально. Но такая сложная форма через объект *IEnumerator* – это для разработчиков компиляторов, а не для прикладных программистов. Поэтому тоже самое можно сделать проще, используя цикл *foreach*.

```
foreach (object o in list) {Console.WriteLine(o); }
```

Эта инструкция делает в точности то, что предыдущая, но выглядит намного читабельнее. Цикл *foreach* можно применить не только к массивам и коллекциям, но к любому объекту, реализующему интерфейс *IEnumerable*.

Интерфейс IDictionary

Последний интерфейс с рисунка – *IDictionary*. Это интерфейс коллекций, которые представляют собой множество пар "ключ – значение". В такой коллекции можно хранить толковый словарь: ключ – слово, значение – его толкование.

```
public interface IDictionary : ICollection, IEnumerable {
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    ICollection Keys { get; }
    ICollection Values { get; }

    object this[object key] { get; set; }

    void Add(object key, object value);
    void Clear();
    bool Contains(object key);
    IDictionaryEnumerator GetEnumerator();
    void Remove(object key); }
```

Благодаря индексатору `object this[object key] { get; set; }` со словарем можно работать как с массивом, пронумерованным не обязательно числами. Например,

```
dict["сидеть"] = "Находиться в положении скорчившись, касаясь задом чего-нибудь твердого."
```

Словарь можно воспринимать как две связанные между собой коллекции: ключей – свойство *Keys*, и значений – свойство *Values.*

```
ICollection Keys { get; }
ICollection Values { get; }
```

Важным условием является то, все ключи словаря должны быть разные, чего нельзя сказать о значениях. Для перебора элементов словаря применяется интерфейс *IDictionaryEnumerator:*

```
public interface IDictionaryEnumerator : IEnumerator {
   DictionaryEntry Entry { get; }
   object Key { get; }
   object Value { get; } }
```

Как видите, помимо членов предка *IEnumerator*, в нем есть:

-структура *Entry* – текущий элемент словаря из двух половинок,

-свойство *Key* – текущий ключ, и

-свойство *Value* – текущее значение.

В соответствие с вариантом задания описать указанные интерфейсы и реализовать их в перечисленных классах. Реализовать события для всех основных действий, допустимых по отношению к конкретному классу. Создать программу на языке C#, демонстрирующую работу реализованных классов, использование интерфейсов, возникновение и обработку событий.

Выполненная работа должна включать исходный текст работоспособной программы и отчет о выполнении работы.

1. Реализовать системы электрических источников и приборов, соединенных между собой через шнуры. В интерфейсах должны быть предусмотрена возможность получения информации о напряжении и максимальной мощности, которую поддерживает элемент. Прибор должен иметь наименование, потребляемую мощность, а источник и провод – списки подключенных приборов.

Интерфейсы:

IElectricSource (источник тока)

IElectricAppliance (электрический прибор)

IElectricWire (электрический шнур)

Классы:

SolarBattery (солнечная батарея)

DieselGenerator (дизельный генератор)

NuclearPowerPlant (атомная электростанция)

Kettle (чайник)

Lathe (токарный станок)

Refrigerator (холодильник)

ElectricPowerStrip (электрический удлинитель)

HighLine (высоковольтная линия)

StepDownTransformer (понижающий трансформатор, должен реализовывать интерфейсы и потребителя и источника тока)

2. Реализовать компоненты компьютерной системы, связанные между собой через определенные интерфейсы. Обеспечить возможность стыковки элементов системы между собой в случае совпадения интерфейсов взаимодействия. Интерфейсы в обязательном порядке поддерживать информацию о максимальной скорости передачи данных и возможность передавать как минимум побайтовые данные.

Интерфейсы:

IUsbBus (шина USB)

ISata (шина SATA)

INetwork (сеть)

IInnerBus (внутренняя шина компьютера)

Классы:

MotherBoard (материнская плата с процессором)

RamMemory (оперативная память)

HardDisk (жесткий диск)

Printer (принтер)

Scanner (сканер изображений)

NetworkCard (сетевая карта)

Keyboard (клавиатура)

3. Реализовать набор коллекций, реализующих стандартные интерфейсы по работе с коллекциями из пространства имен System.Collections.

Интерфейсы:

IEnumerable (последовательность элементов)

ICollection (коллекция)

IList (список)

IDictinary (словарь)

Классы:

List (список)

Queue (очередь)

Dictinary (словарь)

Лабораторная работа №3

Работа с БД

Создание функций для MS SQL Server с использованием платформы .Net Framework.

Данная возможность появилась еще в MS SQL Server 2005, когда стало можно подключать к MS SQL Server свои собственные сборки, созданные для платформы .Net Framework.

Благодаря этому пользователи получили возможность создавать в MS Visual Studio при помощи одного из языков программирования (C#, Visual Basic.Net и др.) объекты, которые впоследствии могут использоваться внутри MS SQL Server.

Разрешается создание следующих объектов:

новые типы данных;

пользовательские функции;

хранимые процедуры;

агрегатные функции;

триггеры.

По умолчанию на MS SQL Server запрещено использование CLR (Common Language Runtime), поэтому требуется явно включить эту опцию на сервере.

Замечание. CLR – это общеязыковая исполняющая среда, которая входит в состав Microsoft .Net Framework и отвечает за выполнение любого кода, созданного для платформы .Net Framework.

C# – объектно-ориентированный язык, разработанный специально для платформы .Net Framework. Однако можно использовать и другие языки программирования для данной платформы.

Рассмотрим процесс создания обычной функции для MS SQL Server на языке C# в MS Visual Studio 2008.

В MS Visual Studio 2008 создадим новый проект (меню **File – New – Project…**)

В диалоге создания проекта выберем нужный тип проекта и укажем имя (рис. 1).

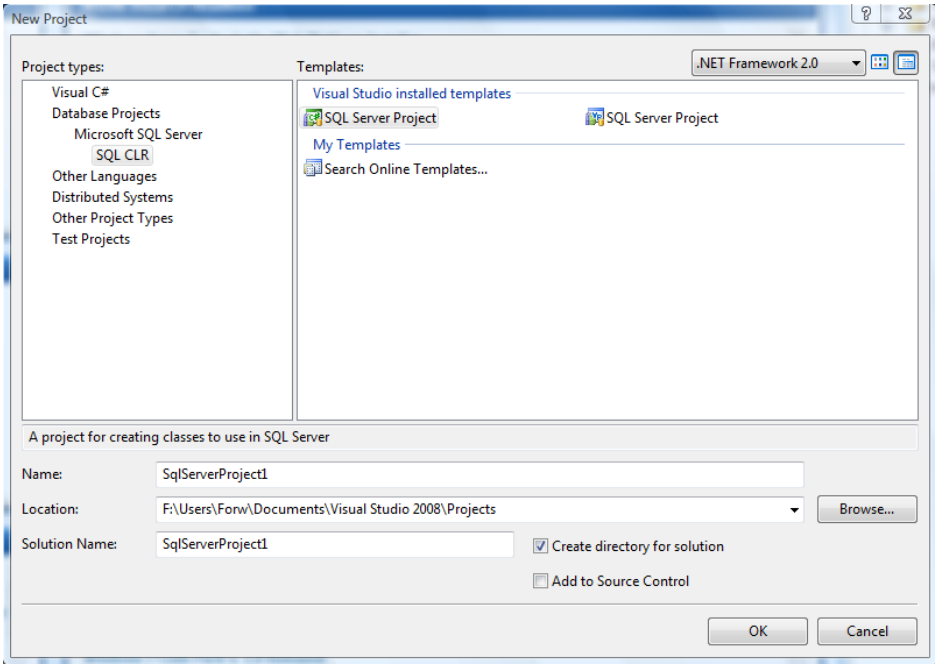


Рис. 1. Создание нового проекта в MS Visual Studio 2008

Далее можно указать сервер и базу данных, которые впоследствии можно будет использовать для отладки кода (см. рис. 2).

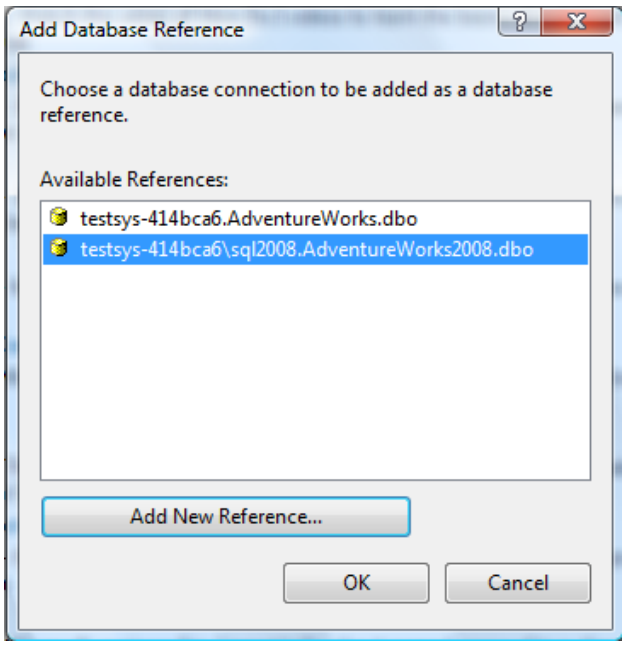


Рис. 2. Выбор сервера и базы данных

В созданный проект добавим новый объект – пользовательскую функцию (см. рис. 3).

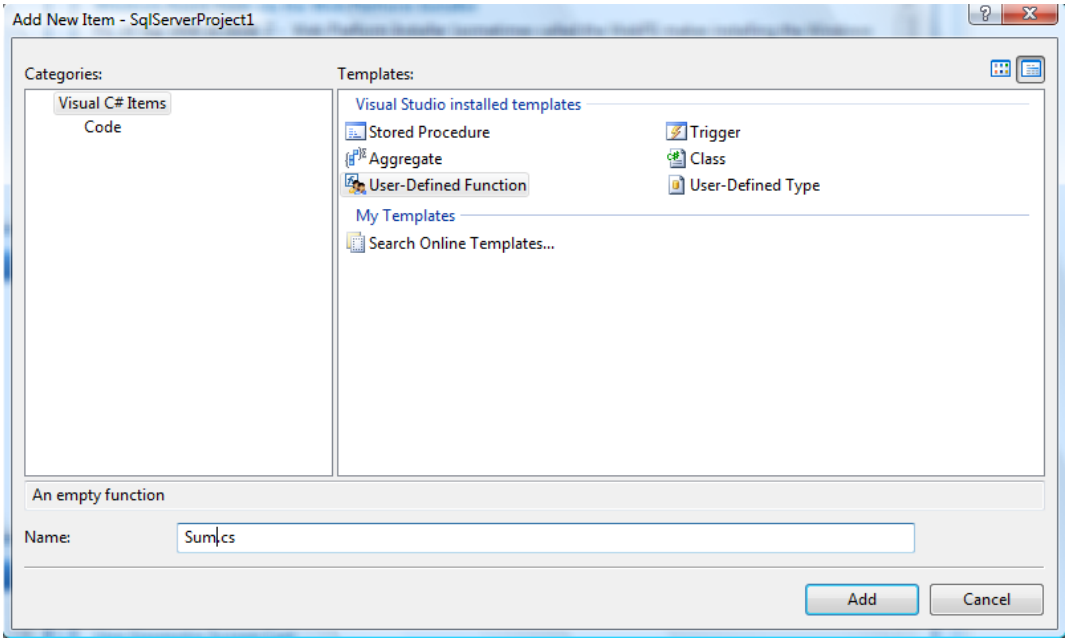


Рис. 3. Добавление функции в проект

Напишем следующий код в этой функции:

using System;

```
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
 [Microsoft.SqlServer.Server.SqlFunction]
 public static SqlInt32 Sum(SqlInt32 a, SqlInt32 b)
 {
        return a + b;
 }
};
```

Обратите внимание, что используются не обычные типы данных, которые есть в C#, а специфичные для MS SQL Server, такие типы имеют префикс «**Sql**»

Зарегистрируем сборку на сервере MS SQL Server. Для этого выполним команду меню **Build** – **Deploy Solution**. При этом будет произведена компиляция сборки, а сама сборка будет помещена на сервер в указанную БД.

Разрешим выполнение CLR кода на сервере:

```
sp_configure 'clr enabled', 1
RECONFIGURE;
```

После этого можно вызвать созданную функцию:

```
SELECT dbo.Sum(10, 20)
```

Аналогичным образом можно создать и агрегатную функцию. Например, агрегатная функция SUM в MS SQL Server работает только с числами, поэтому если возникает необходимость использовать такую функцию для строк (чтобы набор строк объединился в единую строку, а значения были бы разделены запятыми), то это можно реализовать средствами C#.

Лабораторная работа №4

Занесение данных в БД. Работа с DataSet

ADO.NET представляет собой основную модель доступа к данным в платформе .Net Framework. Она не является расширением существовавшей ранее технологии ADO, а реализует новую модель работы с данными. Например, она предполагает отсоединенную модель работы, т.е. подключение с сервером баз данных устанавливается только на момент выполнения запроса, после чего оно разрывается. Ранее подключения с БД поддерживалось на протяжении всего сеанса работы.

ADO.NET включает следующие ключевые компоненты:

наборы данных (DataSet). Представляют собой некоторую часть реальной БД, включая в себя не только таблицы, но и связи между таблицами и ограничения;

провайдеры данных (DataProvider). Благодаря наличию различных провайдеров, технология ADO.NET может работать с различными типами СУБД: MS SQL Server, MS Access, Oracle, а также с любой БД, используя технологию ODBC (Open DataBase Connectivity).

Общая схема работы с использованием ADO.NET:

Создать и установить подключение к серверу. Например, подключимся к локальному серверу к БД ***AdventureWorks***:

```
string connectionString = “Data Source=(local);Initial
Catalog=AdventureWorks; Integrated Security=SSPI;”;
SqlConnection connection = new SqlConnection(connectionString);
connection.Open();
```

Создать команду и выполнить на сервере. Например, получим все записи из таблицы *Orders*:

SqlCommand cmd = new SqlCommand(“SELECT * FROM Orders”, connection);

SqlDataReader reader = cmd.ExecuteReader();

Обработать результаты выполнения команды. Например, выведем содержимое таблицы на консоль:

```
while (reader.Read())
{
 Console.WriteLine(String.Format("{0}, {1}", reader[0], reader[1]));
}
reader.Close();
```

Закрыть соединение:

```
conn.Close();
```

Технология LINQ

LINQ (Language Integrated Query) – проект Microsoft по добавлению синтаксиса языка запросов, напоминающего SQL, в языки программирования платформы .NET Framework.

LINQ позволяет выполнять запросы к объектам находящимся в памяти, в типизированной базе данных и в XML документе. Для этого используются соответствующие технологии: LINQ, DLINQ XLINQ.

Например, создадим массив **arr** и заполним его числами от 0 до 10, а затем при помощи LINQ выберем элементы большие 4 и меньшие 8:

```
int[] arr = { 7, 9, 3, 4, 5, 6, 0, 8, 9, 10 };
var newArray = from i in arr
```

```
where i > 4 && i < 8
select i;
```

В результате переменная **newArray** будет содержать коллекцию целых чисел: 7, 5, 6.

Перед началом работы LINQ с базами данных нам требуется получить описания объектов базы данных в MS Visual Studio. Существуют различные инструменты для генерации сущностей для MS Visual Studio из баз данных. Например, утилита командной строки *SQLMetal*, которую можно найти в папке: C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin.

Пример использования:

```
SQLMetal /server:.\SQL2008 /database:AdventureWorks /pluralize
 /code:AdventureWorks.cs
```

В результате будут сгенерированы объекты на C#, описывающие все объекты БД ***AdventureWorks2008***.

Рассмотрим применение LINQ на практике. Перед выполнением запросов LINQ к базам данных необходимо создать контекст данных:

```
var db = new MyDataContext(@"server=.\SQLEXPRESS; database=my;
 integrated security=SSPI");
if (!db.DatabaseExists())
 db.CreateDatabase();
```

Рассмотрим примеры выполнения стандартных операций над данными при помощи LINQ.

Выборка одного единственного значения:

```
var first = db.Customers.FirstOrDefault(c => c.CustID=="CHIPS");
```

Выборка по условию:

```
where, null, contains & type
var r = new string[] { "WA", "OR" };
var customers = from c in db.Customers
where c is Customer &&
(c.Region==null || r.Contains(c.Region))
select c;
```

Операция соединения таблиц:

```
var labels = (from c in db.Customers join o in db.Orders
on c.CustID equals o.CustID
select new { name = c.ContactName,
address = o.ShipAddress
}).Distinct();
```

Выполнение агрегатных функций:

```
var totals = from c in db.Customers
group c by c.Country into g
select new Summary { Country = g.Key,
CustomerCount = g.Count(),
OrdCount = g.Sum(a=> a.Orders.Count)};
```

Операция вставки новой записи:

```
var customer = new Customer() {
CustID = "CHIPS",
CompanyName = "Mr. Chips" };
db.Customers.InsertOnSubmit(customer);
db.SubmitChanges();
```

Лабораторная работа №5

Создание пакета установки

Microsoft Visual Studio.Net поддерживает самые разные способы развертывания приложения: от самых простых, до сложных, с использованием Windows Installer.

Когда приложение полностью готово, необходимо его полностью подготовить к развертыванию, т.е. к установке на компьютере клиента с минимальными усилиями.

Развертывание приложения с помощью простого копирования

Полное развертывание имеет ряд ограничений:

Все файлы, необходимые для работы приложения должны находиться в этом же каталоге где и приложение (библиотеки, ресурсы, файлы и т.д.).

На целевом компьютере до начала развертывания должна быть установлена среда .NET Framework.

Развертываемое приложение не должно требовать файлов или ресурсов. Кроме тех, которые уже установлены на компьютере.

Обычно такое развертывание осуществляется с помощью команды

X COPY D:\MyApp C:\MyApp/S

Откуда куда

Пуск/Все программы/Стандартные/Командная строка

Создание проекта установочной программы

Для Windows Forms поддерживается два вида установочных проектов:

для приложений (setup project)

для дополнительных модулей Merge Module projects.

Первый применяется для получения дистрибутива, пригодного для развертывания исполняемого приложения, а второй для элементов управления и компонентов, которые не являются отдельными приложениями и не подлежат непосредственному развертыванию.

Для создания дистрибутива готовой программы необходимо добавить к решению проект установочной программы.

Для этого в пункте File проекта необходимо выбрать команду Add Project/New project.

Откроется диалоговое окно Add New Project, содержащее две панели:

левая – панель Project Types

правая – панель Templates

На левой панели выбираем: Setup and Deployment Projects.

А на правой: Setup Wizard.

Затем щелкаем на кнопке ОК.

В результате запускается мастер Setup Wizard и появляется окно с приветствием Setup Wizard(1 of 5). В этом окошке необходимо щелкнуть кнопку Next.

Появляется второе окно Setup Wizard(2 of 5) и страницей Choose a project type, на которой необходимо задать тип проекта с помощью одного из четырех флажков: два флажка для исполняемой сборки

○ Create a Setup for a Windows Application

○ Create a Setup for a Web Application

И два для установки неисполняемой сборки (dll)

○ Create a merge module for Windows Installer

○ Create a dowloadable CAB file.

Обычно (для нас) первый флаг.

Затем Next.

Появится третье окно со страницей: Choose Project Outputs To Include, которая позволяет выбрать файлы решения для включения в проект установочной программы.

На этой странице отображается шесть **флажков**:

□ Documentation Files from…

□ Primary output from…

□ Localized resources from…

□ Debug Symbols from…

□ Content Files from…

□ Source Files from…

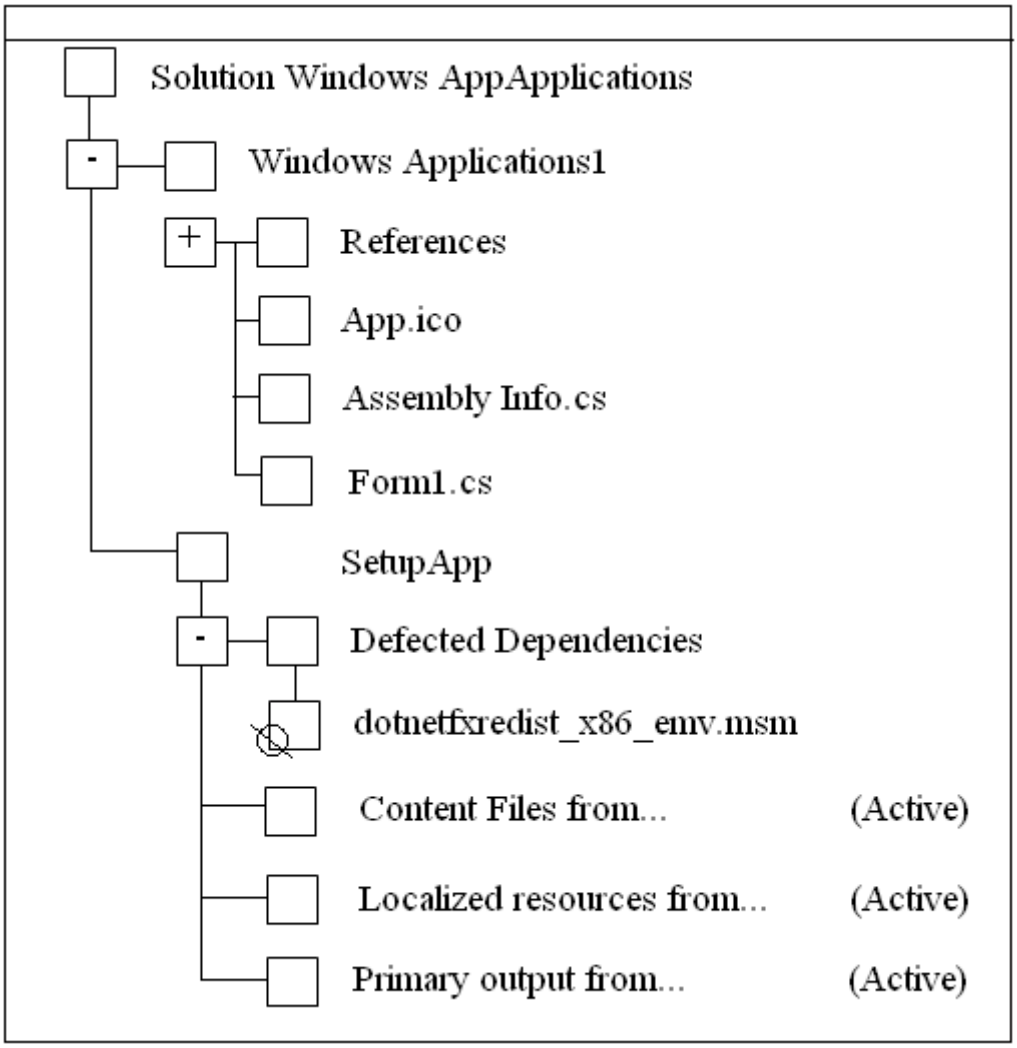
Для того чтобы включить в проект EXE и DLL – файлы необходимо установить флажок Primary output from… . При этом внизу на панели Description появится краткое описание. Обычно еще установлен флажок Content Files from… и флажок Localized resources from…

Можно еще на этапе отладки включить информацию для отладки и исходные тексты, но в конечном варианте их не должно быть.

Затем нажимается кнопка Next, и мастер переходит к 4 шагу из 5 и открывается страница Choose To Include, позволяющая добавить к проекту любые дополнительные файлы, например текстовые файлы, справочные файлы в формате HTML и т.п.

Добавление осуществляется с помощью кнопки Add(если не хотите что-либо добавлять → Next). Появляется страница Create Project последнего шага работы мастера, отображающая информацию о параметрах вашего проекта. Щелкнув кнопку Finish, мы создадим проект установочной программы и добавим его к решению.

Новый проект отобразится в окне Solution Explorer.



Мастер автоматически читает все зависимости вашего проекта и добавит их в папку Detected Dependencies. В нашем случае это зависимость работы нашего проекта от среды .NET Framework. В этом списке (папке) могут присутствовать и другие зависимости. Все они по умолчанию будут помечены знаком (stop), т.е. по умолчанию все найденные зависимости не будут включены в дистрибутив.

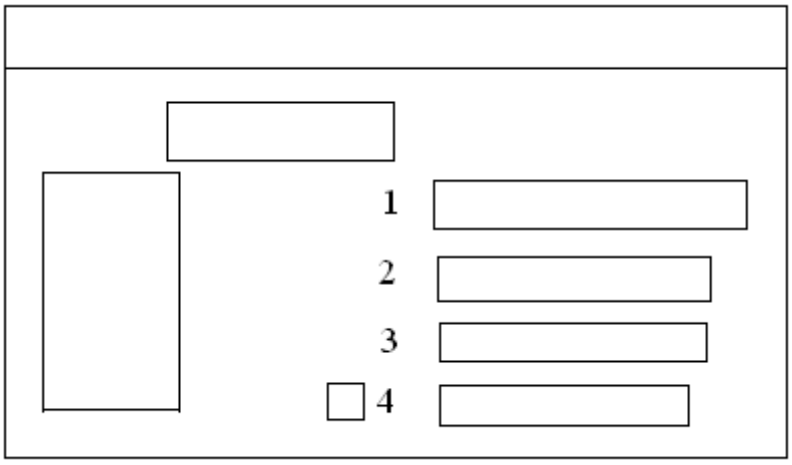
Для включения найденной зависимости в дистрибутив необходимо щелкнуть правой кнопкой на названии файла и сбросить флажок Exclude для

этого файла. Включать dotnet… следует только в том случае, если на целевом компьютере нет среды .NET.

Параметры компоновки установочной программы

Обычно, в результате компиляции установочного проекта, получается, по крайней мере один файл с расширением .msi, в котором хранятся все конфигурационные данные и содержимое, необходимое для установки приложения. Но можно, в зависимости от носителя, на котором предполагается переносить установочные файлы, размещать пакет на нескольких отдельных файлах, а также включать в него установочные файлы Windows Installer, сами они отсутствуют на целевом компьютере.

Для изменения параметров компоновки установочной программы используется диалоговое окно свойств проекта установочной программы. Для этого необходимо в Solution Explorer, щелкнув на имени установочного проекта SetupApp правой кнопкой мыши и выбрать из контекстного меню команду Properties – откроется диалоговое окно свойств проекта.



Output File Name определяет путь и имя, которое будет назначено выходному файлу(установочному) по умолчанию:

`name_cataloge\project_name.msi`

Или `msm` для дополнительных модулей.

Можно изменить каталог с помощью кнопки Browse…

Package Files определяет способ установки выходных файлов решения. По умолчанию все выходные файлы упаковываются в один дистрибутивный файл вместе с установочной программой. Это обеспечивает высокую степень сжатия при минимальной сложности установки.

In Setup file

In Cabinet file(s) – упаковка в несколько CAB файлов, при этом можно укусить размер CAB – файлов в байтах (1,44 для *****). Для этого используется окно CAB size, которое становится доступным, если выбрана опция In Cabinet file.

Если установлен флажок Unlimited, то будет один непрерывный CAB – файл.

Если установлен флажок Custom file, то

□ Unlimited здесь задается max размер

□ Custom [ ] CAB файла.

Опция As loose uncompressed files – позволяет скомпилировать в виде не сжатых файлов.

Bootstrapper – этот параметр определяет необходимо ли включить в дистрибутив пакет Windows Installer. Если выбран этот параметр, то генерируются дополнительные файлы, которые используются для установки на целевом компьютере Windows Installer. Для ОС XP – это не требуется, т.к. он там уже стоит, поэтому для XP можно выбрать None. В противном случае будут сгенерированы следующие файлы:

Setup.exe проверяет установлен ли на целевом компьютере Windows Installer и если нет, то вызывает в зависимости от версии ОС InstMsiA.exe или InstMsiW.exe для установки Windows Installer. После этого запускается Windows Installer для установки приложения, извлекая его из Msi файла.

InstMsiA.exe – для Windows 95/98

InstMsiW.exe – NT/200

Размер каждого из этих файлов при максимальном сжатии около 1,85 МГб.

Setup.exe содержит имя Msi файла, который Setup.exe обрабатывает после проверки Windows Installer.

При выборе параметра Web Bootstrapper открывается диалоговое окно Web Bootstrapper Settings. В нем указывается с помощью свойства: Setup folder URL – Web каталог, в котором находится программа установки и Windows Installer Upgrade folder Url указывается отдельный каталог для загрузки через Web InstMsiA.exe и InstMsiW.exe файлов.

Т.е. программы для установки Windows Installer и самого приложения можно хранить в разных Web каталогах. По умолчанию – в одном.

Compression определяет степень сжатия. None – нет. Optimized For Speed – компромисс между скоростью и объектом. Optimized For Size – max степень сжатия.

На данном этапе можно считать, что все подготовлено для создания дистрибутива, и можно выбрать в окне Solution Explorer правой кнопкой SetupApp и в открывшемся контекстном поле меню выбрать пункт компиляции Build.

Затем сгенерированные файлы можно скопировать на выбранный для распространения носитель.

Конфигурирование проекта установочной программы

В Visual Studio.Net имеется шесть редакторов свойств установочной программы:

Редактор файловой системы (File System Editor) – позволяет выбрать каталог файловой системы целевого компьютера.

Редактор реестра (Registry Editor) создает записи в реестре в процессе установки.

Редактор типов файлов (File Type Editor) связывает типы файлов с обрабатываемыми приложениями.

Редактор пользовательского интерфейса (User Interface Editor) – модифицирует пользовательский интерфейс программы.

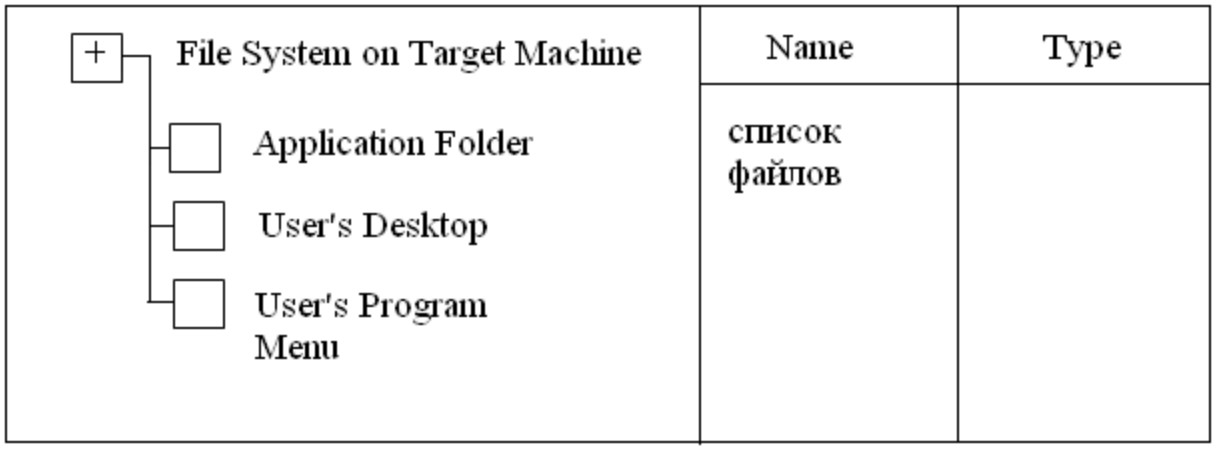
Редактор нестандартных действий (Custom Actions Editor) – залает нестандартные действия, выполняемые установочной программой.

Редактор условий (Launch Conditions Editor) – определяет необходимые условия для начала установки.

Редактор файловой системы

Редактор файловой системы позволяет манипулировать файловой системой целевого компьютера: записывать выходные файлы в различные каталоги, создавать на целевом компьютере не новые каталоги, а также создавать и добавлять ярлыки.

Для запуска редактора файловой системы необходимо выделить SetupApp в окне Solution Explorer и выбрать в меню View команду Editor/File System Editor – окно редактора.



На правой панели отображается список выходных файлов проекта установочной программы, а на левой структура каталогов целевого компьютера.

Первоначально она включает в себя три папки:

Каталог приложения

Рабочий стол

Поле для программ пользователя

Можно добавить свои собственные папки, если щелкнуть левую панель правой кнопкой и выбрать из контекстного меню Add Special Folder, а затем стандартную папку или создать свою. По умолчанию выходные файлы записываются в каталог приложения. Целевой каталог можно изменить, выделив его на правой панели и перетащив его на левую панель в необходимую папку.

Редактор файловой системы позволяет:

Добавить не сжатый файл к дистрибутиву.

Разместить сборки в GAC во время установки приложения.

Добавить в какую-либо папку ярлык:

Для этого на левой панели щелкнуть Application Folder, затем выбрать из списка на правой панели файл, для которого необходимо создать ярлык.

Щелкнуть правой кнопкой файл, для которого необходимо создать ярлык и выбрать из контекстного меню команду Create Shortevt. Ярлык для файла создается и добавляется на панель.

Перетащить ярлык с правой панели в необходимый каталог на левой панели.

Редактор условий установки

Launch Conditions ****** позволяет определять условия, которым целевой компьютер должен удовлетворять для начала установки, например, версия Windows. Кроме того, можно проверять наличие определенных файлов, записей реестра, компонентов, и принимать решения о начале установки по результатам проверки.

Окно редактора поделено на две части: в первой задают объект (файл, раздел реестра или компонент), и во второй – условие, зависящее от наличия этого компонента на целевом компьютере.

Если условия выполняются, то процесс установки продолжается, в противном случае прекращает, и откат.

Лабораторная работа №6

Безопасность Windows-приложений
Небезопасный код

Одним из основных достоинств языка C# является его схема работы с памятью: автоматическое выделение памяти под объекты и автоматическая уборка мусора. При этом невозможно обратиться по несуществующему адресу памяти или выйти за границы массива, что делает программы более надежными и безопасными.

В некоторых случаях возникает необходимость работать с адресами памяти непосредственно, например, при взаимодействии с операционной системой, написании драйверов и др. Такую возможность представляет так называемый небезопасный (unsafe) код.

Небезопасным называется код, выполнение которого среда CLR не контролирует. Он работает напрямую с адресами областей памяти посредством указателей и должен быть явным образом помечен с помощью ключевого слова unsafe, которое определяет так называемый небезопасный контекст выполнения.

Оператор unsafe имеет следующий синтаксис:

unsafe блок операторов

Все операторы, входящие в блок, выполняются в небезопасном контексте.

Компиляция кода, содержащего небезопасные фрагменты, должна производиться с ключом / unsafe. Этот режим можно установить путём настройки среды Visual Studio (Project-Properties-Configuration-Build-Allow Unasfe Code).

Синтаксис указателей

Указатели предназначены для хранения адресов областей памяти. Синтаксис объявления указателя:

Тип данных* переменная;

Тип данных в этом объявлении не может быть классом, но может быть структурой, перечислением, указателем, а так же одним из стандартных типов: sbyte, byte, short, ushort, unt, unit, long, ulong, char, float, double, decimal, bool, void.

Примеры объявления указателей:

```
int* a;                 //указатель на int
Node* pNode;    //указатель на структуру Node
void* p;            //указатель на неопределенный тип
int*[] m;           //одномерный массив указателей на int
int** d;            //указатель на указатель на int
```

В одном операторе можно описать несколько указателей одного и того же типа, например:

```
int*a,b,c;          //три указателя на int
```

Указатели являются отдельной категорией типов данных. Они не наследуются от типа object, и преобразование между типом object и типами указателей невозможно. В частности, для них не выполняется упаковка и распаковка. Однако допускаются преобразования между разными типами указателей, а так же указателями и целыми.

Указатели могут являться локальными переменными, полями, параметрами и возвращаемыми значениями функции. Эти величины подчиняются общим правилам определения области действия и времени жизни.

Указатель void

Указатель типа void означает, что он ссылается на переменную неизвестного типа. Указатель на тип void применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определён (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Указателю на тип void можно присвоить значение указателя любого типа, а также сравнить его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом. Следовательно,

Указатель void может находиться только в левой части оператора присваивания.

Преобразование указателей

Для указателей поддерживаются неявные преобразования из любого типа указателя к типу void*. Любому указателю можно присвоить константу null. Кроме того, допускаются явные преобразования:

между указателями любого типа;

между указателями любого типа и целыми типами (со знаками и без знака).

Корректность преобразований лежит на совести программиста. Преобразования никак не влияют на величины, на которые ссылаются указатели, но при попытке получения значения по указателю несоответствующего типа поведение программы не определенно.

Инициализация указателей

Инициализация указателей может быть осуществлена различными способами.

1 способ. Присваивание указателю адреса существующего объекта:

С помощью операции получения адреса:

int a=5; //целая переменная

int*p=&a; //в указатель записывается адрес a

С помощью значения другого инициализированного указателя:

int*p1=p;// указатель p уже инициализирован

С помощью имени массива, которое трактуется как адрес:

int[]b= new int[]{10,20,30,40}; //массив

fixed (int*p2=b) {...}; //присваивание указателю p2 адреса начала массива

fixed (int*p2=&b[0]){...}; //то же самое

2 способ. Присваивание указателю адреса области памяти в явном виде:

char*p3=(char*)0x12F69E;

здесь 0x12F69E – шестнадцатеричная константа, (Char*) – операция явного приведения типа: константа преобразуется к типу указателя на char.

3 способ. Присваивание нулевого значения:

int*p4=null;

4 способ. Выделение области памяти в стеке и присваивание её адреса указателю:

int*p5=stackalloc int[10];

Здесь операция stackalloc выполняет выделение памяти под 10 величин типа int (массив из 10 элементов) и записывает адрес начала этой области памяти в переменную p5, которая может трактоваться как имя массива, так и указателем.

Основными операциями с указателями являются * и &.

Обе эти операции являются унарными, т. е. имеют один операнд, перед которыми они ставятся. Операция & соответствует операции "взять адрес". Операция * соответствует словам "значение, расположенное по указанному адресу".

Операция * называется разадресацией или разыменованием. Она предназначена для доступа к величине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины.

Пример:

int a=5; //целая переменная

```
int*p=&a;                    //инициализация указателя адресом переменной a
Console.WriteLine (*p);      //операция разадресации, результат равен 5
Console.WriteLine (++(*p));  //результат 6
int[]b=new int[] {10,20,30,40};  //массив
fixed (int*t=b)              //инициализация указателя адресом начала массива
{
int*z=t;                     //инициализация указателя значением другого указателя
for (int i=0;i<b.length;++i)
{
t[i]+=5;                     //доступ к элементу массива (увеличение на 5)
*z+=5;                       //доступ с помощью адресации (увеличение ещё на 5)
++z;                         //инкремент указателя
}
}
```

В приведённом примере доступ к элементам массива выполняется двумя способами: путём индексации указателя t и путём разадресации указателя z, значение которого инкрементируется при каждом проходе цикла для перехода к следующему элементу массива.

Конструкцию *переменная можно использовать в левой части оператора присваивания, так как она определяет адрес области памяти. Для простоты эту конструкцию можно считать именем переменной, на которую ссылается указатель. С ней допустимы всей действия, определённые для величин соответствующего типа.

Оператор fixed фиксирует объект, адрес которого заносится в указатель, для того чтобы его не перемещал сборщик мусора и, таким

образом, указатель остался корректным. Фиксация происходит на время выполнения блока, который записан после круглых скобок.

Арифметические операции с указателями

Арифметические операции с указателями (сложение с целым, разность указателей, инкремент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, элементы которых размещены в памяти последовательно, например, с массивами.

Инкремент перемещает указатель к следующему элементу массива, декремент – к предыдущему.

Разность двух указателей – это разность их значений, деленная на размер типа в байтах. Так, если p и p1 - указатели на элементы одного и того же массива, то операция p - p1 дает такой же результат, как и вычитание индексов соответствующих элементов массива.

К указателям можно прибавлять целое число. Пусть указатель p имеет значение 2000 и указывает на тип byte. Тогда в результате выполнения оператора:

p=p+3;

значение указателя p будет 2003. Если же указатель p1=2000 был бы указателем па float, то после применения оператора:

p1=p1+10;

значение p1 было бы 2040.

Общая формула для вычисления значения указателя после выполнения операции:

p=p+n;

будет иметь вид:

<p>=<p>+n*<количество байт памяти базового типа указателя >

При сложении указателя с целым числом фактически значение указателя изменяется на величину sizeof (тип), где sizeof – операция получения размера величины указанного типа (в байтах). Эта операция применяется только в небезопасном контексте, с её помощью можно получать размеры не только стандартных, но и пользовательских типов данных.

Если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа.

Другие арифметические операции над указателями запрещены, например нельзя сложить два указателя, умножить указатель на число и т. д.

Указатели можно сравнивать. Применимы все шесть операций:

<,>,<=, >=, = ,= =, !=

Сравнение p<g означает, что адрес, находящийся в p, меньше адреса, находящегося в g. Если p и g указывают на элементы одного массива, то индекс элемента, на который указывает p, меньше индекса массива, на который указывает g.

Пример применения арифметических операций с указателями:

```
//демонстрация операции инкремента
int * p;
int a = 5;
p = &a;
Console.WriteLine("{0:X}",(uint)p);// вывод значения указателя(адреса), //тип uint-беззнаковый целый тип в unicode-формате(16-ное число)
p++;//операция инкремента. Значение указателя(адрес) увеличивается на 4 //байта
Console.Write("{0:X}", (uint)p);
// аналогичные действия, только применимые для типа long:
long *p1;
```

```
p1++;//значение указателя(адрес) увеличится на 8 байт!
//демонстрация операции разности указателей
int []a=new int[]{1,2,3};
        fixed (int* p1 = a)
        {
           int* p2 = p1;
           p2 ++;
            Console.WriteLine(p2-p1);
        }
//демонстрация операции сложения указателя с целым числом
int *p3=p2+2; // к указателю прибавили 2 элемента!
Console.WriteLine(*p3);//значение по указателю равно 3
```

Приоритетность выполнений операций с указателями

При записи выражений с указателями следует обращать внимание на приоритеты операций. В качестве примера рассмотрим последовательность действий, заданную в операторе: *p++=10;

Поскольку инкремент постфиксный, он выполняется после выполнения операции присваивания. Таким образом, сначала по адресу, записанному в указателе p, будет записано значение 10, а затем указатель увеличится на количество байтов, соответствующее его типу. То же самое можно записать подробнее:

```
*p=10;
p++;
```

Выражение (*p)++, напротив, инкрементирует значение, на которое ссылается указатель.

В следующем примере каждый байт беззнакового целого числа x выводится на консоль с помощью указателя t:

```
uint x = 0xAB10234F;
byte*t=(byte*)&x;
```

for (int i=0;i<4;++i)

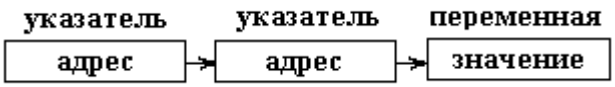
Console.Write (“{0:X}”, *t++); //результат 4F 23 10 AB

Первоначально указатель t был установлен на младший байт переменной x.

Указатель на указатель

В языке C# возможна также ситуация, когда указатель указывает на указатель. В этом случае описание будет иметь следующий вид:

int **point;



Переменная point имеет тип указатель на указатель на int. Чтобы получить целочисленное значение переменной, на которую указывает point следует использовать:

**point;

Пример:

```
int i=7;
int *p;
        int **pp;
p=&i;
pp=&p;
Console.WriteLine("i={0:X}  p= {1:X}  pp= {2:X}",i,(uint)p,(uint)pp);
Console.WriteLine("i={0}  *p= {1}  **pp= {2}",i,*p,**pp);
++*p;
Console.WriteLine("i={0}  *p= {1}  **pp= {2}", i, *p, **pp);
**pp=12;
Console.WriteLine("i={0}  *p= {1}  **pp= {2}", i, *p, **pp);
```

Операция stackalloc

Операция stackalloc позволяет выделять память в стеке под заданное количество величин заданного типа:

stackalloc тип [количество]

Количество задаётся целочисленным выражением. Если памяти недостаточно, генерируется исключение System.StackOverflowException. Выделеная память ничем не инициализируется и автоматически освобождается при завершении блока, содержащего эту операцию. Пример выделения память под пять элементов типа int и заполнения числами от 0 до 4:

```
int*p=stackalloc int [5];
for (int i=0; i<5; ++i)
{
        p[i]=i;
        Console.Write (p[i] +"  “);                    //Результат 0 1 2 3 4
}
```

Написать программу на C# по вариантам с использованием небезопасного кода.

Примечание. Размерности массивов задаются именованными константами.

Вариант 1.

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

1) сумму отрицательных элементов массива;

2) произведение элементов массива, расположенных между максимальным и ми-нимальным элементами.

Упорядочить элементы массива по возрастанию.

Вариант 2.

В одномерном массиве, состоящем из вещественных элементов, вычислить:

1) сумму положительных элементов массива;

2) произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.

Упорядочить элементы массива по убыванию.

Вариант 3.

В одномерном массиве, состоящем из вещественных элементов, вычислить:

1) максимальный элемент массива;

2) сумму элементов массива, расположенных до последнего положительного элемента.

Сжать массив, удалив из него все элементы, модуль которых находится в интер-вале . Освободившиеся в конце массива элементы заполнить нулями.

Вариант 4.

В одномерном массиве, состоящем из вещественных элементов, вычислить:

1) минимальный элемент массива;

2) сумму элементов массива, расположенных между первым и последним положительными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, равные нулю, а потом — все остальные.

Вариант 5.

В одномерном массиве, состоящем из целых элементов, вычислить:

1) номер максимального элемента массива;

2) произведение элементов массива, расположенных между первым и вторым нулевыми элементами.

Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в нечетных позициях, а во второй половине — элементы, стоявшие в четных позициях.

Вариант 6.

В одномерном массиве, состоящем из вещественных элементов, вычислить:

1) номер минимального элемента массива;

2) сумму элементов массива, расположенных между первым и вторым отрицательными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, модуль которых не превышает 1, а потом — все остальные.

Вариант 7.

В одномерном массиве, состоящем из N вещественных элементов, вычислить:

1) максимальный по модулю элемент массива;

2) сумму элементов массива, расположенных между первым и вторым положи-тельными элементами. .

Преобразовать массив таким образом, чтобы элементы, равные нулю, располагались после всех остальных.

Вариант 8.

В одномерном массиве, состоящем из целых элементов, вычислить:

1) минимальный по модулю элемент массива;

2) сумму модулей элементов массива, расположенных после первого элемента, равного нулю.

Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в четных позициях, а во второй половине — элементы, стоявшие в нечетных позициях.

Вариант 9.

В одномерном массиве, состоящем из η вещественных элементов, вычислить:

1) номер минимального по модулю элемента массива;

2) сумму модулей элементов массива, расположенных после первого отрицательного элемента.

Сжать массив, удалив из него все элементы, величина которых находится в интервале. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 10.

В одномерном массиве, состоящем из η вещественных элементов, вычислить:

1) номер максимального по модулю элемента массива;

2) сумму элементов массива, расположенных после первого положительного элемента.

Преобразовать массив таким образом, чтобы сначала располагались все элемен-ты, целая часть которых лежит в интервале [a , b], а потом — все остальные.

Основная литература:

1. Введение в программирование на языке Visual C#: Учебное пособие / С.Р. Гуриков. - М.: Форум: НИЦ ИНФРА-М, 2013. - 448 с.: 70x100 1/16. - http://znanium.com/bookread2.php?book=404441

2. Осипов, Н.А. Разработка Windows приложений на C#. [Электронный ресурс] — Электрон. дан. — СПб. : НИУ ИТМО, 2012. — 74 с. — Режим доступа: https://e.lanbook.com/reader/book/40725/#50

3. Технология разработки программного обеспечения : учеб. пособие / Л.Г. Гагарина, Е.В. Кокорева, Б.Д. Виснадул ; под ред. Л.Г. Гагариной. — М. : ИД «ФОРУМ» : ИНФРА-М, 2017. — 400 с. — Режим доступа: http://znanium.com/bookread2.php?book=768473

Дополнительная литература

4. Гуриков С.Р. Введение в программирование на языке Visual C# : учеб. пособие / С.Р. Гуриков. — М. : ФОРУМ : ИНФРА-М, 2017. — 447 с. — Режим доступа: http://znanium.com/bookread2.php?book=752394

5. Объектно-ориентированное программирование с примерами на C#: Учебное пособие / Хорев П.Б. - М.: Форум, НИЦ ИНФРА-М, 2016. - 200 с. — Режим доступа: http://znanium.com/bookread2.php?book=529350

6. Ступина, А. А. Технология надежностного программирования задач автоматизации управления в технических системах [Электронный ресурс] : монография / А. А. Ступина, С. Н. Ежеманская. - Красноярск : Сиб. федер. ун-т, 2011. - 164 с. — Режим доступа: http://znanium.com/bookread2.php?book=442655