**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**
**Федеральное государственное автономное образовательное учреждение
высшего образования**
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
**к практическим занятиям
по дисциплине
«Объектно-ориентированное программирование»**

для направления подготовки
09.03.02 Информационные системы и технологии
Направленность (профиль)
«Информационные системы и технологии в бизнесе»

Часть 1

Невинномысск, 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ ... 2
Практическое занятие 1. Определение класса ... 3
Практическое задание №1 ... 11
Практическое занятие 2. Классы и функции ... 12
Практическое задание №2 ... 29
Практическое занятие 3. Наследование и полиморфизм ... 30
Практическое задание №3 ... 43
Практическое занятие 4. Массивы объектов, указатели и ссылки ... 45
Практическое задание №4 ... 50
Практическое занятие 5. Шаблоны и исключительные ситуации ... 52
Практическое задание №5 ... 59
Практическое занятие 6. Потоки и классы ввода/вывода ... 61
Практическое задание №6 ... 70
Практическое занятие 7. Статические и константные члены, локальные классы ... 72
Практическое задание №7 ... 78
Практическое занятие №8. Работа с файлами. ... 80
ЛИТЕРАТУРА ... 92

ВВЕДЕНИЕ

Работа на практических занятиях по учебной дисциплине «Объектно-ориентированное программирование» предполагает изучение студентами основ объектно-ориентированного программирования.

Изучение дисциплины предполагает формирование компетенции

Код, формулировка компетенции	Код, формулировка индикатора
ПК-4 Способен разработать архитектуру ИС	ИД-1$_{\text{ПК-4}}$ Осуществляет разработку стратегии развития информационных технологий инфраструктуры предприятия и управления ее реализацией
	ИД-2$_{\text{ПК-4}}$ Осуществляет разработку архитектуры ИТ и ИС инфраструктуры предприятия
	ИД-3$_{\text{ПК-4}}$ Осуществляет обоснование архитектуры ИС

Выполнение практических заданий включает:

1. Изучение студентами необходимого теоретического материала по теме лабораторной работы.
2. Постановку задачи в соответствии с темой лабораторной работы и согласование ее с руководителем.
3. Построение алгоритма решения задачи и его документирование в разделе «Краткие теоретические сведения» отчета.
4. Выполнение задания.
5. Подготовку отчета о выполненной работе и его защиту.

Структура отчета по проделанной работе:

1. Тема.
2. Цель работы.
3. Постановка задачи.
4. Ход выполнения работы.
5. Блок-схема или псевдокод алгоритма решения задачи.
6. Текст программы.
7. Распечатка результатов.
8. Выводы.

Практическое занятие 1. Определение класса

Простейшее определение класса без наследования имеет вид:

```
class имя_класса  {
// по умолчанию раздел private – частные члены класса
     public:  //открытые функции и переменные класса
    };
```

Определение класса соответствует введению нового типа данных, а понятие переменной данного типа – понятию объекта (экземпляра) класса. Список членов класса включает определение данных и функций, те из них, чьи объявления находятся в описании класса, называются функциями-членами класса. В ООП для таких функций используется термин "методы класса". Классы могут также содержать определения функций, тело которых помещается в определение класса (inline-функции). Для инициализации/уничтожения объектов используются специальные функции-конструкторы с тем же именем, что и класс, и деструкторы, с именем класса, перед которым стоит символ "~".

Переменные, объявленные в разделе класса по умолчанию как частные (private), имеют область видимости в пределах класса. Их можно сделать видимыми вне класса, если поставить перед началом объявления слово public.

Обычно переменные в классе объявляются private-переменными, а функции видимыми (public). Открытые функции-члены класса имеют доступ ко всем закрытым данным класса, через них возможен доступ к этим данным.

Классами в С++ являются также структуры (struct) и объединения (union). Отличием структуры и объединения от класса является то, что их члены по умолчанию открытые (public), а не закрытые, как у класса. Это обеспечивает преемственность с языком С. Кроме того, структуры и объединения не могут наследоваться и наследовать.

В следующей программе определяется простейший класс Strtype, членами которого являются массив str типа char и функции set(), show(), get().

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
class Strtype{
     char str[80]; //private
     public:
         void set (char *);    //задать str
        void show();           //вернуть str
        char* get();           //вернуть str
            };                 //конец     определения
класса
void Strtype::set(char *s)// определение метода set()
```

```
    {
         strcpy(str,s);       //копирование s в str
    }
    void Strtype::show()            //   определение   метода
show()
    {
         cout<<str<<endl;
    }
    char * Strtype::get()     // определение метода get()
    {
         return str;
    }
    int main()
    {
         Strtype obstr;             //объявление объекта
         obstr.set("String example"); //вызов       метода
     set()
         obstr.show();                  //вызов       метода
show()
         cout<<obstr.get()<<endl;
         while(!kbhit());//задержка   выхода   до   нажатия
клавиши
         return 0;
    }
```

Вывод: String example
String example

Массив член-класса str – является частным (private), доступ к нему возможен только через функции-члены класса. Такое объединение в классе сокрытых данных и открытых функций и есть инкапсуляция. Здесь obstr – объект данного класса. Вызов функции осуществляется из объекта добавлением к имени объекта имени функции, через точку или –>, если используется указатель на объект.

Приведем примеры нескольких объявлений объектов:

```
Strtype a; //объект a
Strtype x[100]; //массив объектов
Strtype *p; //указатель на объект
p=new Strtype; // создание динамического объекта
a.set("строка"); // вызов функций
x[i].set("строка");
p->set("строка");
```

Оператор расширения области видимости “::” указывает доступ к элементам класса. Например, Strtype::set(char *s) означает принадлежность функции set(char *s) области видимости класса Strtype. Кроме этого оператор “::” используется для доступа к данным класса (оператор вида Strtype::count),

для указания внешней или глобальной области видимости переменной, скрытой локальным контекстом (оператор вида ::globalName).

Следующая программа демонстрирует создание класса Stack на основе динамического массива. Для инициализации объекта класса используется метод Stack() – конструктор класса.

```
#include <iostream.h>
#include <conio.h>
#define SIZE 10
// объявление класса Stack для символов
class Stack{
     char *stck;      // содержит стек
     int tos;         // индекс вершины стека
public:
     Stack();             //конструктор
     ~Stack(){delete [] stck;}//деструктор
     void push(char ch); // помещает в стек символ
     char pop(); // выталкивает из стека символ
};
// инициализация стека
Stack::Stack()
{
     stck=new char[SIZE];//динамический массив
     tos=0;
     cout << "работа конструктора …  \n";
}
// помещение символа в стек
void Stack::push(char ch)
{
     if (tos==SIZE)
     {
         cout << "стек полон";
         return;
     }
     stck[tos]=ch;
     tos++;
}
// выталкивание символа из стека
char Stack::pop()
{
     if (tos==0)
     {
         cout << "стек пуст";
         return 0;
     }
     tos--;
```

```
        return stck[tos];
    }
    int main()
    {
    /*образование   двух   автоматически   инициализируемых
стеков */
        Stack s1, s2; //вызов конструктора для s1 и s2
        int i;
        s1.push('a');
        s2.push('x');
        s1.push('b');
        s2.push('y');
        s1.push('c');
        s2.push('z');
        for(i=0;i<3;i++)
            cout<<"символ из s1:"<<s1.pop() << "\n";
        for(i=0;i<3;i++)
            cout<<"символ из s2:"<<s2.pop() << "\n";
        cout.flush();
        while(!kbhit());//задержка
        return 0;
    }
```

Вывод:

```
Работа конструктора …
Работа конструктора …
Символ из s1:c
Символ из s1:b
Символ из s1:a
Символ из s2:z
Символ из s1:y
Символ из s1:x
```

Конструктор Stack() вызывается автоматически при создании объектов класса s1,s2 и выполняет инициализацию объектов, состоящую из выделения памяти для динамического массива и установки указателя на вершину стека в нуль. Конструктор может иметь аргументы, но не имеет возвращаемого значения и может быть перегружаемым.

Деструктор ~Stack() выполняет действия необходимые для корректного завершения работы с объектом, а именно, в деструкторе может освобождаться динамически выделенная память, закрываться соединения с файлами, и др. Он не имеет аргументов. Именем деструктора является имя класса, перед которым стоит знак “~” – тильда.

Рассмотрим еще один пример класса, реализующего динамический односвязный список:

```
#include <iostream.h>
#include <conio.h>
struct Node            // объявление класса Node
{
     int info;         //информационное поле
     Node* next;       //    указатель    на    следующий
элемент
};
class List                 // объявление класса List
{
     Node* top;        // указатель на начало списка
public:
     List()            // конструктор
     {
         top=0;
         cout<<"\nkonstructor:\n";
     };
     ~List()           // деструктор
     {
         release();
         cout<<"\ndestructor:\n";
     }
     void push(int);  // добавление элемента
     void del()        // удаление элемента
     {
         Node* temp = top;
         top = top->next;
         delete temp;
     }
     void show();
     void release();// удаление всех элементов
};
void List::push(int i)
{
     Node* temp = new Node;
     temp->info = i;
     temp->next =top;
     top=temp;
}
void List::show()
{
     Node* temp=top;
     while (temp!=0)
     {
         cout<<temp->info<<"->";
```

```
            temp=temp->next;}
            cout<<endl;
        }


    void List::release()
    {
        while (top!=0)del();
    }
    int main()
    {
        List *p          ;//объявление указателя на List
        List st;//объявление объекта класса List
        int n = 0;
        cout << "Input an integer until 999: ";
        do      //добавление в список, пока не введено
999
        {
            cin >> n;
            st.push(n);
        } while(n != 999);
        st.show();
        st.del();
        p=&st;
        p->show();
        while(!kbhit());
        return 0;
    }
```

Необходимо отметить, что некоторый класс ClassA может использоваться до его объявления. В этом случае перед использованием класса ClassA нужно поместить его неполное объявление:

```
class ClassA;
```

Важнейшим принципом ООП является наследование. Класс, который наследуется, называется базовым, а наследуемый – производным. В следующем примере класс Derived наследует компоненты класса Base, точнее компоненты раздела public, которые остаются открытыми, и компоненты раздела protected (защищенный), которые остаются закрытыми. Компоненты раздела private, также наследуются, но являются не доступными напрямую для производного класса. Производному классу доступны все данные и методы базового класса, наследуемые из разделов public и protected. Для переопределенных методов в производном классе действует принцип полиморфизма, который будет рассмотрен ниже. Объекту базового класса можно присвоить объект

производного, указателю на базовый класс – значение указателя на производный класс. В этом случае через указатель на базовый класс можно получить доступ только к полям и функциям базового класса. Для доступа к полям и функциям порожденного класса следует привести (преобразовать) ссылку на базовый класс к ссылке на порожденный класс.

```
#include <iostream.h>
#include <conio.h>
class Base                //    определение    базового
класса
{
     int i;          //private по умолчанию
protected:
     int k;
public:
     Base()
     {
         i=0;
         k=1;
     }
     void set_i(int n);   // установка i
     int get_i()      // возврат i
     {
         return i;
     }
     void show()
     {
         cout<<i<<"  "<<k<<endl;
     }
};                   //конец Base
class Derived : public Base        //       производный
класс
{
     int j;
public:
     void set_j(int n);
     int mul();   //умножение i на k базового класса
                  //и на j производного
     }; //конец Derived
//установка значения i в базовом классе

void Base::set_i(int n)
{
     i = n;
}
```

```
//установка значения j в производном классе

void Derived::set_j(int n)
{
     j = n;
}

//возврат i*k из Base умноженного на  j из Derived
int Derived::mul()
{
/*производный  класс  наследует  функции-члены  базового
класса*/
     return  j  *  get_i()*k;//вызов  get_i()  базового
класса
}
int main()
{
     Derived ob;
     ob.set_i(10);              //загрузка i в Base
     ob.set_j(4);          // загрузка j в Derived
     cout << ob.mul()<<endl;   //вывод числа 40
     ob.show();            //вывод i и k, 10 1
     Base bob=ob;          //присваивание      объекта
ссылке
                           //на базовой тип
     cout<<bob.get_i();        // вывод i
     cout.flush();
     while (!kbhit());
     return 0;
}
```

Переменная i недоступна в производном классе, переменная k доступна, поскольку находится в разделе protected. В производном классе наследуются также функции get_i(), set_i() и show() класса Base из раздела public. Функция show() позволяет получить доступ из производного класса к закрытой переменной i производного класса. В результате выводится 40 10 1 10.

Принцип полиморфизма, состоящий в перегрузке методов, объявленных в различных классах с одним и тем же именем и списком параметров, будет рассмотрен в следующих темах.

Практическое задание №1

Цель работы: Изучить определение классов в языке С++.

Постановка задачи: Создать класс, отражающий структуру данных согласно выбранному варианту задания. Класс должен содержать методы для чтения, установки и отображения своих данных. Реализовать класс List для управления динамическим списков из элементов созданного класса.

Варианты заданий

№ варианта	Структура данных
1	Дата
2	Время
3	Книга
4	Ф.И.О.
5	Адрес
6	Студент
7	Автомобиль
8	Компьютер
9	Предприятие
0	Страна

Практическое занятие 2. Классы и функции

При реализации класса используются функции-члены класса (методы класса), функции-“друзья” класса, конструкторы, деструкторы, функции-операторы. Функции-члены класса объявляются внутри класса. Определение функции обычно помещается вне класса. При этом перед именем функции помещается операция доступа к области видимости имя_класса::. Таким образом, определение функции-члена класса имеет вид:

```
тип имя_класса:: имя_функции (список аргументов) {  }
```

Определение функции может находиться и внутри класса вслед за его объявлением. Такие функции называются inline-функциями. Рассмотрим пример:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#define SIZE 255
class Strtype
{
     char *p;
     int len;
public:
     Strtype()    //  инициализация  объекта,  inline
конструктор
     {
          p=new char [SIZE];
          if(!p)
          {
               cout << "ошибка выделения памяти\n";
               exit(1);
          }
          *p='\0';
          len=0;
     }
     ~Strtype(); //деструктор
     void set(char *ptr);
     char *get()
     {
          return p;
     }
     void show();
```

```
};
// освобождение памяти при удалении объекта строка
inline Strtype::~Strtype()//inline-деструктор
{
     cout << "освобождение p\n";
     delete [] p;
}
void Strtype::set(char *ptr)
{
     if(strlen(ptr) >= SIZE)
     {
         cout << "строка слишком велика \n";
         return;
     }
     strcpy(p, ptr);
     len = strlen(p);
}
void Strtype::show()
{
     cout << p << " длина : " << len<< "\n";
}
int main()
{
 {   Strtype s1,s2;
     s1.set("This is a test");
     s2.set("Second test");
     s1.show();
     s2.show();
     cout<<s2.get()<<endl;
 }
     while (!kbhit());
     return 0;
}
```

В результате будет выведено:

```
This is a test длина: 14
Second test длина: 11
Second test
Освобождение p
Освобождение p
```

Конструктор Strtype() и функция get() являются inline-функциями. Деструктор ~Strtype() также является inline-функцией, хотя и определяется вне класса. Ключевое слово inline содержит указание компилятору создать код

функции, подставляемый в точку вызова. При этом время вызова сокращается, хотя код может увеличиться.

Вызов функций-членов класса осуществляется одним из двух способов:

- имя_объекта.имя_функции(аргументы);
- указатель_на_объект -> имя_функции(аргументы);

Еще раз обращаем внимание на то, что при определении функции- члена класса перед именем функции стоит имя класса, а при вызове –имя объекта класса.

Указатель this

Обращение к данным класса осуществляется с помощью функций, которые могут вызываться различными объектами класса. Возникает вопрос: откуда функция "знает", какой объект его вызвал и какие данные при этом должны быть использованы? Например, в предыдущем примере вызовы функции s1.show() и s2.show() приводят к выводу различных для s1и s2 значений строк p. Это происходит вследствие того, что каждый объект имеет скрытый указатель this на самого себя, объявляемый неявно. Значением этого указателя, который автоматически передается функциям-членам класса при их вызове, является адрес начала объекта. В результате функция show() в реальности содержит инструкцию:

```
cout << this->p << " длина : " << this->len<< "\n";
```

При этом *this представляет сам объект, this->имя_члена – ссылка на данные объекта класса. Рассмотрим пример:

```
#include<iostream.h>
#include<conio.h>
class MyClass
{
     int x,y;
     public:
     void set(int xx, int yy)      //устанавливает
данные
     {
         x=xx;    //равносильно this->x=xx;this->y=yy;
         y=yy;
     }
     MyClass f(MyClass &); //возвращает объект
     MyClass *ff()//возвращает указатель на вызвавший
объект
     {
         x=y=100;
         return this;
```

```
        }
        void display()
        {
            cout<<x<<'\t'<<y<<endl;
        }
    };
    MyClass MyClass::f(MyClass &M)
    {
        x+=M.x;
        y+=M.y;
        return *this;
    }
    int main()
    {
        MyClass a, b;
        a.set(10,20);
        b.ff()->display();   //результат 100,100
        b.display();     //результат 100,100
        a.display();     //результат 10,20
        a.f(b).display();    //результат 110,120
        a.display();     //результат 110,120
        while (!kbhit());
        return 0;
    }
```

Так как функция b.ff() возвращает указатель this на объект b, то можно вызвать функцию display() как this->display(). При этом выводятся поля данных объекта b(100,100). При вызове a.f(b).display() функция f(b) возвращает значение вызвавшего ее объекта a как *this. Затем вызывается функция display() как a.display().

Отметим, что в функцию лучше передавать не значение объекта, а ссылку на него. Это дает экономию стековой памяти, поскольку объект не копируется, и более безопасно для объектов, использующих динамическую память. Уничтожение копий аргументов при выходе из функции не может разрушить такие объекты при освобождении динамической памяти.

Функции-друзья класса, объявляемые со спецификатором friend, указатель this не содержат. Объекты, с которыми работают такие функции, должны передаваться в качестве их аргументов.

Конструкторы и деструкторы

Конструктор – это функция-член класса, которая вызывается автоматически для создания и инициализации экземпляра класса. Известно, что экземпляры структур можно инициализировать при объявлении:

```
struct Student
{
     int semhours;          //public по умолчанию
     char subj;
}
Student s={0,"c" };        //объявление                 и
инициализация
```

При использовании класса приложение не имеет доступа к его защищенным элементам, поэтому для класса подобную инициализацию выполнить нельзя. Для этой цели используются специальная функция-член класса, инициализирующая экземпляр класса и указанные переменные при создании объекта. Подобная функция вызывается всегда при создании объекта класса. Это и есть конструктор – функция-член класса, которая имеет то же имя, что и класс. Конструктор может быть подставляемой (inline) и неподставляемой функциями. Рассмотрим пример:

```
#include<iostream.h>
class Student
{
     int semhours;
     char subj;
public:
     Student() //inline-конструктор 1 без параметров
     {
          semhours=0;
          subj='A';
     }
     Student(int, char);//объявление
                        //конструктора 2 с параметрами
};
Student::Student(int hours,char g)//конструктор 2
{
     semhours=hours;
     subj=g;
}
int main()
{
     Student s(100,'A'); //конструктор 2
     Student s1[5]; //конструктор1 вызывается 5 раз
     return 0;
}
```

Там, где находятся объявления s и s1, компилятор помещает вызов соответствующего конструктора Student().

Конструктор не имеет типа возвращаемого значения, хотя может иметь аргументы и быть перегружаемым. Он вызывается автоматически при создании объекта, выполнении оператора new или копировании объекта. Если конструктор отсутствует в классе, компилятор C++ генерирует конструктор по умолчанию.

Деструктор вызывается автоматически при уничтожении объекта, и имеет то же имя, что и класс, но перед ним стоит символ "~". Деструктор можно вызывать явно в отличие от конструктора. Конструкторы и деструкторы не наследуются, хотя производный класс может вызывать конструктор базового класса.

В следующем примере используется конструктор копирования, который необходимо вводить из-за проблемы, связанной с динамической памятью. Когда объект передается в функцию по значению, то создается его копия, при этом конструктор не вызывается. При выходе из функции объект уничтожается и вызывается деструктор, освобождающий его динамическую память. В этом случае исходный объект, использующий ту же динамическую память, что и копия, может быть поврежден. Аналогично, если функция возвращает объект, содержащий динамические переменные, при выходе из функции копия этого объекта разрушается вызовом деструктора. Возвращаемый объект при этом также может быть разрушен. Равным образом это повторяется и при инициализации объявляемого объекта вида:

```
Class_type B=A;
```

При этом происходит побитовое копирование объекта A в объект B. Однако для массива в объекте B память динамически не выделяется, а происходит только копирование указателя на массив. Если затем объект A, содержащий динамический массив, разрушается вызовом деструктора, то объект B также разрушается. Поэтому при таком объявлении должен вызваться конструктор копирования.

Общая форма конструктора копирования имеет вид:

```
имя_класса (const имя_класса &ob)
{
/*выделение памяти и копирование в нее ob*/
}
```

Здесь ob является ссылкой на объект в правой части инициализации.

Рассмотрим пример использования конструктора копирования при создании безопасного динамического массива.

```
/*создается класс "безопасный" массив. Когда один
объект-массив используется для инициализации другого, для
выделения памяти вызывается конструктор копирования*/
```

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
class Array
{
     int *p;
     int size;
public:
     Array (int s)
     {
         cout << "constructor1 \n";
         size=s;
         p=new int[size];
         if(!p) exit(1);
     }
     ~Array()
     {
         delete [ ] p;
     }
     Array(const Array &a);   //конструктор
копирования
     void put(int i, int j)
     {
         if(i>=0 && i<size) p[i]=j;
     }
     int get(int i)
     {
         return p[i];
     }
};
/* конструктор копирования. Память выделяется для
копии, адрес памяти передается p.*/
Array:: Array(const Array &a)
{
     p=new int[a.size];   //выделение памяти для
копии
     if(!p) exit(1);
     for(int i=0;i<a.size;i++)
     p[i]=a.p[i];//копирование содержимого в память
для копии
     cout << "constructorcopy2\n";
}
int main()
{
     Array y(5);      //вызов обычного конструктора
```

```
        for(int i=0;i<5;i++)
        {   //помещение в массив нескольких значений
            y.put(i, i+5);
            cout << y.get(i);    //вывод на экран num
        }
        cout << "\n";
        Array x=y;
    /*создание   массива   x   и   инициализация,   вызов
конструктора копирования */
    // другой способ вызова - объявление: Array x(y);
        y=x;//конструктор не вызывается !!!
        while(!kbhit());
        return 0;
    }
```

Результат:

```
constructor1
56789
constructorcpy2
```

Объект y используется при инициализации объекта x с помощью конструктора копирования. При этом выделяется динамическая память, адрес которой в x.p, и производится копирование y в x. После этого y и x не разделяют одну и ту же динамическую память. Если же присваивание осуществляется не при инициализации, то конструктор копирования не вызывается, а происходит побитовое копирование объекта. В результате двумя объектами используется одна и та же динамическая память. Так происходит в случае присваивания y=x в примере.

В следующем примере конструктор копирования вызывается при передаче объекта-строки в качестве аргумента функции show():

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
class Strtype
{
     char *p;
     int l;
public:
     Strtype(char *s='\0')    //конструктор
     {
         l=strlen(s);
         p=new char[l+1];
         if(!p)
```

```
            {
                cout<<     "ошибка     при     выделении
памяти\n";exit(1);
            }
            strcpy(p, s);
            cout<<"constr1\n";
    }
        Strtype(const      Strtype      &o)//конструктор
копирования
        {
            l=strlen(o.p);
            p=new char[l+1];//выделение памяти для новой
копии
            if(!p)
            {
                cout << "ошибка памяти\n";
                exit(1);
            }
            strcpy(p, o.p);  //передача строки в копию
            cout<<"constrcpy\n";
        }
        ~Strtype()       //деструктор
        {
            delete [ ] p;
            cout<<"destr\n";
        }
        char *get()
        {
            return p;
        }
    };
    void show(Strtype x)
    {
        char *s;
        s=x.get();
        cout << s << "\n";
    }
    int main()
    {
        Strtype a("Hello"), b("There");//constr1,constr1
        show(a);//конструктор                  копирования
constrcpy,Hello,destr
        show(b);//конструктор                  копирования
constrcpy,There,destr
        while(!kbhit());
```

```
    return 0;
}
```

Когда функция show() завершается и x выходит из области видимости, то освобождается память x.p, однако не та, которая используется передаваемым в функцию объектом.

Организация внешнего доступа к компонентам класса. Функции “друзья” класса.

В языке С++ одна и та же функция не может быть компонентом двух разных классов. Чтобы предоставить функции возможность выполнения действий над различными классами можно определить обычную функцию языка С++ и предоставить ей право доступа к элементам класса типа private, protected. Для этого нужно в описании класса поместить заголовок функции, перед которым поставить ключевое слово friend. Рассмотрим пример:

```
//функция sameColor() сравнивает цвет прямоугольника и линии
class Line;//неполное объявление класса
class Box
{
    int color;       // цвет прямоугольника
    int upx, upy;        // левый верхний угол
    int lowx, lowy;  //правый нижний угол
public:
    // объявление friend-функции
    friend int sameColor(line l, box b);
    void setColor(int c);
    void defBox(int x1, int y1, int x2, int y2);
};
class Line
{
    int color;
    int startx, starty;
    int endx,endy;
public:
    friend int sameColor(Line l, Box b);
    void setColor(int c);
    void defLine(int x, int y, int l);
};
int sameColor(Line l, Box b)
{
    if(l.color==b.color) return 1;
    return 0;
```

```
}
```

Если не использовать friend-функцию, то необходимо сначала вернуть цвета объектов, а затем написать функцию их сравнения. В приведенном примере класс Box использует класс Line, объявленный позже. Поэтому перед объявлением класса Box ставится предварительное неполное объявление вида:

```
class имя_класса;
```

Хотя дружественная функция "знает" элементы класса, доступ к ним возможен только через объект класса, т.к. указатель this ей не передается. В следующем примере дружественная функция использует указатель на класс в качестве параметра и для доступа к элементам i и k через указатель. Прямой доступ к ним невозможен.

```
class X
{
      int i,k;
public:
      void memFunc(int);
      friend void frFunc(X*, int);
};
void X::memFunc(int a)
{
      i=a;     //прямой доступ
      k=a;
}
void fr_func(x *xptr, int a)
{
//доступ   к   i   и   k   через   указатель,   ошибка,   если
записать к=a
      xptr->i=a;
      xptr->k=a;
}
int main()
{
      X xobj;//объявление объекта
      frFunc(&xobj, 6);//вызов дружественной функции
      xobj.memFunc(6);//вызов члена класса
      return 0;
}
```

Разрешается объявлять функции-члены класса дружественными для другого класса. В следующем примере функция memF() член класса X получает доступ к защищенным полям класса Y.

```
class Y;
class X
{//...
     void memF(Y ob);
};
class Y
{//…
     friend void X::memF(Y ob);
};
```

Операторы-функции

Операторы-функции используются для введения операций, связанных с символами:

```
+ , - , * , / , % , ^ , & , | , ~ , ! , = , < , >
,+= , [ ] , -> , ( ) , new, delete.
```

Оператор-функция является членом класса или дружественной классу. Общая форма оператора-функции члена класса:

```
возвращаемый_тип имя_класса::operator#(список_аргум)
{ /*выполняемые действия */ }
```

После этого вместо operator#(a,b) можно писать a#b. Здесь символ # представляет один из введенных выше символов. В качестве примера можно привести операции >>, <<, перегружаемые для ввода-вывода. Отметим, что при перегрузке нельзя менять приоритет операторов и число операндов. Если оператор-функция, являющаяся членом класса перегружает бинарный оператор, у функции будет только один параметр-объект, находящийся справа от знака оператора. Объект слева вызывает оператор-функцию и передается неявно с помощью указателя this. Например:

```
// перегрузка  +, = и ++ для класса  coord.
#include <iostream.h>
#include <conio.h>
class Coord
{
     int x, y; // значения координат
public:
     Coord(int i=0, int j=0) { x = i; y = j; }
     void getXY(int &i, int &j) { i = x; j = y; }
     Coord operator+(Coord ob2);
     Coord operator=(Coord ob2);
     Coord operator++(); // префиксная форма
```

```
};
Coord Coord::operator+(Coord ob2) //перегрузка  +
{
      Coord temp;
      temp.x = x + ob2.x; // temp.x=this->x+ob2.x
      temp.y = y + ob2.y; // temp.y=this->y+ob2.y
      return temp;
}
Coord Coord::operator=(Coord ob2) //перегрузка =
{
      x = ob2.x;   // this->x=ob2.x
      y = ob2.y;   // this->y=ob2.y
      return *this;    //возвращение объекта,
                   //которому присвоено значение
}
Coord  Coord::operator++()  //перегрузка  ++,  унарный
оператор
{
      x++;
      y++;
      return *this;
}
int main()
{
      Coord o1(10, 10), o2(5, 3), o3;
      int x, y;
      o3  =  o1  +  o2;  //сложение  двух  объектов.  Вызов
operator+()
      o3.getXY(x, y);
      cout << "(o1 + o2) X: " << x << ", Y: " << y <<
"\n";
      o3 = o1; //присваивание объектов
      o3.getXY(x, y);
      cout << "(o3 = o1) X: " << x << ", Y: " << y <<
"\n";
      ++o1; //инкрементация объекта
      o1.getXY(x, y);
      cout << "(++o1 )     X: " << x << ", Y: " << y <<
"\n";
      while(!kbhit());
      return 0;
}
```

Результат:

```
(o1+o2) X:15 ,y:13
(o3=o1) X:10, y:10
```

```
 (++o1)  X:11, Y:11
```

При перегрузке унарного оператора ++ единственный параметр-объект передается через указатель this.

В следующем примере вводятся класс “множество” и операции && – пересечения, << – вывод множества.

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
class Set
{
     char  *pl; //указатель  на  множество  элементов
типа char
public:
     Set(char *pi)//конструктор
     {
          pi=new char[strlen(pl)+1];
          strcpy(pi,pl);
     }
     Set  &operator  &&(Set  &);  //перегрузка  &&-
пересечение
     //перегрузка<<
     friend  ostream  &operator<<(ostream  &stream,Set
&ob);
     ~Set(){delete [] pi;}//деструктор
};
Set& Set::operator &&(Set &s) // пересечение A=A^B
{
     int  l=0;
     for (int j=0;pi[j]!=0;j++)
          for (int k=0;s.pi[k]!=0;k++)
               if (pi[j]==s.pi[k])
               {
                    pi[l]=pi[j];
                    l++;
                    break;
               }
     pi[l]=0;
     return *this;
}
ostream &operator<<(ostream &stream, Set &ob)
{
     stream << ob.pi << '\n'; /*перегрузка вывода */
     return stream;
}
```

```
int main()
{
	Set s1("1f2bg5e6"),s2("abcdef");
	Set s3=s2;
	cout<<(s1&&s2)<<endl;//результат fbe
	cout<<s3<<endl;//результат abcdef
	while(!kbhit());
	return 0;
}
```

Оператор присваивания

Когда одному объекту присваивается значение другого, происходит его копирование. Если побитовое копирование нежелательно из-за того, что копии объектов ссылаются на одну и ту же динамическую память, можно использовать собственную функцию копирования operator=(). Рассмотрим пример:

```
Strtype &Strtype::operator=(const Strtype &ob)
{
	if(&ob==this) return *this;
	if(l < ob.l)
	{ //требуется выделение дополнительной памяти
		delete[ ] p;
		p = new char [ob.l+1];
		if(!p)
		{
			cout << "ошибка выделения памяти \n";
			exit(1);
		}
	}
	l = ob.l;
	strcpy(p, ob.p);
	return *this;
}
```

Здесь operator=() возвращает ссылку на объект. Параметром также является ссылка, что запрещает создание копии объекта, стоящего справа от операции присваивания. Добавление этого оператора в класс позволяет копировать объекты с помощью оператора присваивания, который не наследуется и должен быть членом класса.

Использование дружественных операторов-функций

В дружественную функцию не передается указатель this, поэтому при перегрузке для унарного оператора в оператор-функцию передается один параметр, для бинарного – два. Оператор присваивания не может быть дружественной функцией, а только членом класса. В следующем примере рассматривается использование дружественных оператор-функций +, *, /, <<, >> для класса Complex.

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
class Complex
{
      double re, im;
public:
      Complex (double  r=0,double i=0)//конструктор
      {
            re=r;
            im=i;
      }
      friend Complex operator+(Complex, Complex);
      friend Complex operator*(Complex, Complex);
      friend Complex operator/(Complex, Complex);
      friend  ostream  &  operator<<(ostream  &stream,
Complex ob);
      friend  istream  &  operator>>(istream  &stream,
Complex ob);
};
      ostream &operator<<(ostream &stream, Complex ob)
      {
            stream  <<"("<<  ob.re  <<  ",  "  <<  ob.im
<<")"<< '\n';
            return stream;
      }
istream &operator>>(istream &stream, Complex &ob)
{
      stream >> ob.re >> ob.im;
      return stream;
}
Complex operator+(Complex a1, Complex a2)
{
      return Complex(a1.re+a2.re,a1.im+a2.im);
}
Complex operator*(Complex a1, Complex a2)
{
      return Complex(a1.re*a2.re-a1.im*a2.im,
                  a1.re*a2.im+a1.im*a2.re);
```

```
}
Complex operator/(Complex a1, Complex a2)
{
      double r=a2.re,i=a2.im;
      double tr=r*r+i*i;
      return Complex((a1.re*r+a2.im*i)/tr,
                  (a1.im*r-a1.re*i)/tr);
}
int main()
{
      Complex a(1.,2.),b(2.,2.),c;
      c=a+b;
      cout<<c<<a*b<<a/b<<endl;
      cout<<"Enter complex number:";
      cin>>c;
      cout<<c;
      while(!kbhit());
      return 0;
}
```

Практическое задание №2

Цель работы: Научиться работать с функциями-членами класса, конструкторами, деструкторами, дружественными классами и перегруженными операторами.

Постановка задачи: Создать класс согласно выбранному варианту задания. Реализовать процедуры ввода и отображения данных.

Варианты заданий

№ варианта	Задание
1	Однонаправленный список целых чисел с использованием inline-функций.
2	Стек из строк с использованием inline конструкторов и деструкторов.
3	Однонаправленный список объектов типа «Книга» с использованием указателя this.
4	Стек из объектов типа «Адрес» с использованием указателя this.
5	Создать «безопасный массив» из объектов типа «Автомобиль» с использованием конструкторов и деструкторов.
6	Однонаправленный список объектов типа «Дата» с использованием конструкторов копирования.
7	Создать объекты типа «Вектор» и «Матрица». Реализовать в одном их этих классов дружественную функции для умножения вектора на матрицу.
8	Однонаправленный список из объектов типа «Дата». Для класса «Дата» перегрузить операторы сложения и вычитания.
9	Однонаправленный список из объектов типа «Время». Для класса «Время» перегрузить оператор присваивания.
0	Для классов «Вектор» и «Матрица» перегрузить операции сложения, вычитания и умножения с использованием дружественных операторов-функций.

Практическое занятие 3. Наследование и полиморфизм

При реализации наследования производному классу (потомку) передаются свойства базового класса, объявленные в разделах public и protected. Члены раздела protected являются частными для базового и производного классов. При наследовании класса может объявляться спецификатор доступа к членам базового класса.

```
class имя_класса: доступ имя_базового_класса
{
//члены класса
}
```

Спецификатор доступа при наследовании может принимать одно из трех значений: public (по умолчанию при наследовании структур), private (по умолчанию при наследовании классов) и protected. В случае принятия спецификатором значения public все члены разделов public и protected базового класса становятся членами разделов public и protected производного класса. Члены раздела private (частные) недоступны для производного класса. Если доступ имеет значение private, то все члены разделов public и protected становятся членами раздела private производного класса. Когда же доступ принимает значение protected, то все члены разделов public, protected переходят в раздел protected производного класса. Рассмотрим пример:

```
#include <iostream.h>
#include <conio.h>
class X
{
protected: //обязательно для наследования
     int i, j;
public:
     void getIJ()
     {
         cout << "Enter two number: ";
         cin >> i >> j;
     }
     void showIJ()
     {
         cout << i << " " << j << "\n";
     }
};
class Y : public X
{
     /* в классе Y, i и j - защищенные данные из X */
     int k;
```

```
public:
      int getK()
      {
          return k;
      }
      void makeK()
      {
          k = i*j;
      }
};
/*Z имеет доступ к i и j из X, но не к частному  k
класса Y */
class Z : public Y
{
public:
      void f()
      {
          i = 2;
          j = 3;
      }// i и j доступны отсюда
};
int main()
{
      Y v1;
      Z v2;
      v1.getIJ();
      v1.showIJ();
      v1.makeK();
      cout << v1.getK()<<"\n";
      v2.f();
      v2.showIJ();
      while(!kbhit());
      return 0;
}
```

Вывод: Enter two numbers:5 6

```
5 6
30
2 3
```

Конструкторы и деструкторы производных классов

Конструкторы и деструкторы базовых классов не наследуются производными классами, однако они вызываются при создании объектов этих классов. При этом конструкторы выполняются в порядке наследования, а деструкторы – в обратном порядке. Для передачи аргументов конструктору

базового класса при использовании производного класса применяется следующий синтаксис:

```
конструктор_производного_класса(арг):Base(арг)
{
     //тело конструктора производного класса
}
```

Рассмотрим пример:

```
#include <iostream.h>
#include <conio.h>
class Base
{
     int i;
public:
     Base(int n)
     {
         cout << "constructor1 \n";
         i = n;
     }
     ~Base()
     {
         cout<<"destructor1 \n";
     }
     void showi()
     {
         cout << i << '\n';
     }
};
class Derived : public Base
{
     int j;
public:
     Derived(int n):Base(n+5)
     /*передача аргумента в базовый класс */
     {
         cout << "constructor2  \n";
         j = n;
     }
     ~Derived()
     {
         cout<<"destructor2\n";
     }
     void showj()
```

```
        {
            cout << j << '\n';
        }
    };
    int main()
    {
        {
            Derived o(3);
            /*фигурные скобки добавлены, чтобы вывести
сообщения             деструкторов */
            o.showi();
            o.showj();
        }
        while(!kbhit());
        return 0;
    }
```

Вывод:

```
constructor1
constructor2
8
3
destructor2
destructor1
```

Множественное наследование

Один класс может наследовать атрибуты двух и более базовых классов, которые перечисляются после двоеточия через запятую. Если базовые классы содержат конструкторы, то они вызываются поочередно в порядке перечисления.

```
#include <iostream.h>
#include <conio.h>
class X
{
protected:
    int a;
public:
    X()
    {
        a = 10;
        cout << "инициализация X\n";
    }
    ~X()
    {
        cout << "деинициализация X\n";
```

```
	}
};
class Y
{
protected:
	int b;
public:
	Y()
	{
		cout << "инициализация Y\n";
		b = 20;
	}
	~Y()
	{
		cout << "деинициализация Y\n";
	}
};
// Z наследует как  X, так и Y
class Z : public X, public Y
{
public:
	Z()
	{
		cout << "инициализация Z\n";
	}
	~Z()
	{
		cout << "деинициализация Z\n";
	}
	int make_ab() { return a*b; }
};
int main()
{
	{
		Z ob;
		cout << ob.make_ab();
	}
	while(!kbhit());
	return 0;
}
```

Результат:

```
инициализация X
инициализация Y
инициализация Z
200
```

```
деинициализация Z
деинициализация Y
деинициализация X
```

В C++ указатели и ссылки на базовый класс могут быть использованы для ссылок на объект производного класса. Если параметр функции является ссылкой на базовый класс, то аргументом функции может быть как объект базового класса, так и объект производного. Однако через указатель или ссылку базового класса, которые в действительности ссылаются на объект производного класса, можно получить доступ только к тем объектам (полям и функциям) производного класса, которые были унаследованы от базового класса. В то же время указатель или ссылку производного класса нельзя использовать для доступа к объекту базового класса. В этом случае необходимо применить явное преобразование типов, используя операцию (тип).

Виртуальные функции и полиморфизм

Механизм виртуальных функций в ООП используется для реализации полиморфизма: создания метода, предназначенного для работы с различными объектами из иерархии наследования за счет механизма позднего (динамического) связывания. Виртуальные функции объявляются в базовом и производном классах с ключевым словом virtual, имеют одинаковое имя, список аргументов и возвращаемое значение. Метод, объявленный в базовом классе как виртуальный, будет виртуальным во всех производных классах, где он встретится, даже если слово virtual отсутствует.

В отличие от механизма перегрузки функций (функции с одним и тем же именем, но с различными типами аргументов считаются разными) виртуальные функции объявляются в порожденных классах с тем же именем, возвращаемым значением и типом аргументов. Если различны типы аргументов, виртуальный механизм игнорируется. Тип возвращаемого значения переопределить нельзя.

Основная идея в использовании виртуальных функций состоит в следующем: такая функция может быть объявлена в базовом классе, а затем переопределена в каждом производном классе. Каждый объект класса, управляемого из базового класса с виртуальными функциями, содержит указатель на таблицу vmtbl (virtual method table) с адресами виртуальных функций. Эти адреса устанавливаются в адреса нужных для данного объекта функций во время выполнения. При этом доступ через указатель на объект базового класса осуществляется к виртуальной функции из базового класса, а доступ через указатель на объект производного класса – на функцию из этого же класса. То же происходит при передаче функции объекта производного класса, если аргумент объявлен как базовый класс.

Оба варианта рассмотрены в следующем примере:

```
/*здесь указатель на базовый класс и ссылка используются для доступа к виртуальной функции */
#include <iostream.h>
```

```
#include <conio.h>
class Base
{
public:
     virtual void who()   //  определение  виртуальной
функции
     {
          cout << "Base\n";
     }
};
class First : public Base
{
public:
     void who()   //  определение  who()  относительно
First
     {
          cout << "First\n";
     }
};
class Second : public Base
{
public:
     void  who()//  определение  who()  относительно
Second
     {
          cout << "Second\n";
     }
};
/* параметр ссылка на объект базового класса */
void show_who(Base &r)
{
     r.who();
}
int main()
{
     Base base_obj;
     Base* pb;
     pb=&base_obj;
     pb->who();       //base_obj.who()
     First first_obj;
     pb=&first_obj;
     pb->who();    //first_obj.who()
     Second second_obj;
     pb=&second_obj ;
     pb->who();         //second_obj.who()
```

```
      show_who(base_obj); // доступ к Base's who()
      show_who(first_obj); // доступ к First's who()
      show_who(second_obj); // доступ к Second's who()
      while(!kbhit());
      return 0;
}
```

Вывод:

```
Base
First
Second
Base
First
Second
```

В следующем примере рассматривается реализация стека и очереди на основе односвязного списка, при этом очередь и стек являются потомками класса List:

```
/*классы, наследование и виртуальные функции
создание класса родовой список для целых*/
#include <iostream.h>
#include <stdlib.h>
class List
{
public:
     List *head; //указатель на начало списка
     List *tail;//указатель на конец списка
     List *next;//указатель на следующий элемент
     int num;          //число для хранения
     List ()
     {
         head = tail = next = NULL;
     }
     /*абстрактная базовая виртуальная функция*/
     virtual void store(int i) = 0;
     /*абстрактная базовая виртуальная функция*/
     virtual int retrieve() = 0;
 };
 //создание типа очередь на базе списка
class Queue : public List
{
public:
     void store(int i);
     int retrieve();
```

```
     queue operator+(int i)
     {
          store(i);
          return *this;
     }
     /* перегрузка постфиксного инкремента */
     int operator --(int unused)
     {
          return retrieve();
     }
};
void Queue::store(int i)
{
     List *item;
     item = new Queue;
     if(!item)
     {
          cout << "ошибка выделения памяти\n";
          exit(1);
     }
     item -> num = i;
     //добавление в конец списка
     if(tail) tail -> next = item;
     tail = item;
     item -> next = NULL;
     if(!head) head = tail;
}
int Queue::retrieve()
{
     int i;
     List *p;
     if(!head) {cout << "список пуст\n";return 0;   }
     //удаление из начала списка
     i = head -> num;
     p = head;
     head = head -> next;
     delete p;
     return i;
}
class Stack : public List
{/*создание класса стек на базе списка */
public:
     void store(int i);
     int retrieve();
     Stack operator+(int i)
```

```
    {
        store(i);
        return *this;
    }
    int operator --(int unused)
    {
        return retrieve();
    }
};
void Stack::store(int i)
{
    List *item;
    item = new Stack;
    if(!item) {
        cout << "ошибка выделения памяти\n";
        exit(1);
    }
    item -> num = i;
    //добавление в начало списка, как в стеке
    if(head) item -> next = head;
    head = item;
    if(!tail) tail = head;
}
int Stack::retrieve()
{
    int i;
    List *p;
    if(!head)
    {
        cout << "список пуст\n";
        return 0;
    }
    //удаление из начала списка
    i = head -> num;
    p = head;
    head = head -> next;
    delete p;
    return i;
 }
int main()
{
    List *p;
    //демонстрация очереди
    Queue q_ob;
    p = &q_ob; //указывает на очередь
```

```
        q_ob + 1;
        q_ob + 2;
        q_ob + 3;
        cout << "очередь : ";
        cout << q_ob --;// вывод 1
        cout << q_ob --;// вывод 2
        cout << q_ob --;// вывод 3
        cout << '\n';
        //демострация стека
        Stack s_ob;
        p = &s_ob; //указывает на стек
        s_ob + 1;
        s_ob + 2;
        s_ob + 3;
        cout << "стек: ";
        cout << s_ob --;// вывод 3
        cout << s_ob --;// вывод 2
        cout << s_ob --;// вывод 1
        cout << '\n';
        return 0;
    }
```

Если конструкторы не могут быть виртуальными, деструкторы могут (для полиморфных объектов их рекомендуется объявлять в базовых классах как виртуальные).

Абстрактные классы и чисто виртуальные функции

Абстрактные классы – это классы, содержащие чисто виртуальные функции, которые при объявлении в классе приравниваются к нулю. Абстрактные классы используются только для наследования, так как объекты таких классов не могут быть созданы.

```
#include<iostream.h>
class Shape       //абстрактный
{
     int xb,yb,xc,yc;
public:
     Shape(int hb,int vb,int hc,int vc)://конструктор
     /* применяется список инициализаторов */
     xb(hb),yb(vb),xc(hc),yc(vc){}
     virtual void move(int,int)=0;//чисто виртуальная
функция
```

```
        virtual    void    copy(int,int)=0;//    чисто
виртуальная функция
        virtual void rotate(int)=0;// чисто виртуальная
функция
    };
```

При реализации конструктора использован список инициализаторов, который помещается после двоеточия следующего за заголовком конструктора. Для каждого поля в скобках указывается инициализирующее значение (может быть выражением). Это единственный способ инициализации полей констант, полей ссылок и полей объектов, у которых есть только конструкторы с параметрами. В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам.

Абстрактные базовые классы используются для создания общего для множества иерархических классов интерфейса. В следующем примере рассматриваются два класса с общим интерфейсом, через который осуществляется вызов функций с помощью указателей на соответствующую vtbl – таблицу виртуальных методов.

```
    #include <iostream.h>
    #define interface struct
    interface IX
    {
        virtual void _stdcall FX1()=0;
        virtual void _stdcall FX2()=0;
    };
    class CA:public IX
    {
        /*наследование структуры  */
    public:
        virtual                void                 _stdcall
FX1(){cout<<"CA::FX1"<<endl;}
        virtual                void                 _stdcall
FX2(){cout<<"CA::FX2"<<endl;}
    };
    class CB:public IX
    {
    public:
        virtual                void                 _stdcall
FX1(){cout<<"CB::FX1"<<endl;}
        virtual                void                 _stdcall
FX2(){cout<<"CB::FX2"<<endl;}
    };
    void ff(IX* pix)
    {
```

```
    pix->FX1();
    pix->FX2();
}
int main()
{
    CA* pA=new CA;//создание экземпляра CA
    CB* pB=new CB;//создание экземпляра CB
    IX* pix=pA;
    ff(pix);//вызов методов CA
    pix=pB;
    ff(pix);//вызов методов CB
    return 0;
}
```

Практическое задание №3

Цель работы: Изучить объектно-ориентированные концепции наследования и полиморфизма в языке программирования С++.

Постановка задачи: Создать набор классов согласно выбранному варианту задания. Реализовать процедуры ввода и отображения данных.

Варианты заданий

№ варианта	Задание
1	Создать базовый класс «Фигура» и несколько производных от него классов: «Точка», «Квадрат», «Круг» и т.д. Классы должны содержать приватные, защищенные и общедоступные члены данных.
2	Создать базовый класс «Транспорт» и несколько производных от него классов: «Автомобиль», «Самолет», «Поезд» и т.д. Классы должны содержать приватные, защищенные и общедоступные члены данных.
3	Создать базовый класс «Фигура» с использованием конструктора с аргументами и деструктора. Создать несколько производных от него классов: «Точка», «Квадрат», «Круг» и т.д.
4	Создать базовый класс «Транспорт» с использованием конструктора с аргументами и деструктора. Создать несколько производных от него классов: «Автомобиль», «Самолет», «Поезд» и т.д.
5	Создать базовый класс «Фигура» и производные от него классы «Точка», «Квадрат», «Круг». Создать базовый класс «Оформление» и производные от него классы «Цвет» и «Стиль». На основе этих классов создать набор различных классов с использованием множественного наследования.
6	Создать базовый класс «Транспорт» и производные от него классы «Автомобиль», «Поезд», «Самолет». Создать базовый класс «Тип» и производные от него классы «Грузовой» и «Пассажирский». На основе этих классов создать набор различных классов с использованием множественного наследования.
7	Выполнить задание 1 с использованием виртуальных

	функций для методов ввода и вывода данных.
8	Выполнить задание 2 с использованием виртуальных функций для методов ввода и вывода данных.
9	Выполнить задание 1, при условии, что базовый класс должен быть абстрактным.
0	Выполнить задание 2, при условии, что базовый класс должен быть абстрактным.

Практическое занятие 4. Массивы объектов, указатели и ссылки

Массивы объектов создаются так же, как и массивы переменных. Если класс содержит конструктор, массив может быть инициализирован, причем конструктор вызывается столько раз, сколько элементов содержит массив.

```
#include<iostream.h>
class Sample
{
     int a;
public:
     Sample(int n)
     {
         a=n;
         cout<<"constructor\n";
     }
     int getA(){return a;}
};
int main()
{
     Sample ob[4]={1,2,3,4};
     Sample *pob=ob;
     int i;
     for(i=0;i<4;i++)
         cout<<ob[i].getA ()<< ' ';
     cout<<"\n";
     cout<<pob->getA ();
     return 0;
}
```

Программа выведет 1, 2, 3, 4, конструктор вызывается четыре раза. Затем еще раз будет выведено 1 с помощью указателя pob на первый элемент массива. Список инициализации – это сокращение общей конструкции:

```
Sample
ob[4]={Sample(1),Sample(2),Sample(3),Sample(4)};
```

Такая конструкция становится основной, если конструктор имеет два и более аргумента. Например:

```
Sample                                          ob[4]=
 {Sample(1,2),Sample(3,4),Sample(5,6),
              Sample(7,8)};
```

При создании динамических объектов используются указатели на объект оператор new, который вызывает конструктор и производит инициализацию. Для разрушения динамического объекта применяется оператор delete, который может помещаться в деструкторе. Например:

```
#include <iostream.h>
class Samp
{
     int i, j;
public:
     Samp()
     {
         cout<<”конструктор2\n”;
     }
     Samp(int a,int b)
     {
         i=a;j=b;
         cout<<”конструктор1\n”;
     }
     void setIJ(int a, int b)
     {
         i = a;
         j = b;
     }
     ~Samp() { cout << "удаление...\n"; }
     int get() { return i*j; }
};
int main()
{
     Samp *p01;
     int i;
     p01 = new Samp(6,5);
     Samp *p02;
     p02 = new Samp[3];
     if(!p01||!p02)
     {
         cout << "ошибка выделения памяти\n";
         return 1;
     }
     for(i=0; i<3; i++)
     {
         p02[i].setIJ(i, i);
         cout     <<     "p02["     <<     i     <<
"]="<<p02[i].get()<<”\n”;
     }
```

```
    cout << p01->get() << "\n";
    delete p01;
    delete [ ] p02;
    return 0;
}
```

Результат:

```
конструктор1
конструктор2
конструктор2
конструктор2
p02[0]=0
p02[1]=1
p02[2]=4
30
удаление...
удаление...
удаление...
удаление...
```

Деструктор вызывается 4 раза: по одному разу на каждый элемент массива и один раз для объекта p01.

Ссылки

Ссылка является скрытым константным указателем и работает как другое имя переменной. Ее можно использовать для передачи аргументов в функцию и возврата значений обратно. При передаче объекта через ссылку в функцию сообщается адрес объекта, а его копия не делается. Это уменьшает вероятность ошибок, связанных с выделением динамической памяти и вызовом деструктора. Аналогично, при возврате ссылки на объект из функции также не делается копия объекта.

При передаче функции объекта в качестве параметра может возникнуть ошибка из-за разрушения деструктором копии объекта, которая должна быть исправлена созданием конструктора копирования. В этой ситуации лучше использовать функцию, возвращающую ссылку на объект. Например:

```
// защищенный двумерный массив
#include <iostream.h>
#include <stdlib.h>
class Array
{
    int isize, jsize;
    int *p;
public:
    Array(int i, int j)
```

```
        {
            p = new int [ i * j ];
            if(!p)
            {
                cout << "Ошибка выделения памяти\n";
                exit(1);
            }
            isize = i;
            jsize = j;
        }
        int &put(int i, int j);
        int get(int i, int j);
    };
    /* возврат ссылки на элемент массива, в который
необходимо выполнить запись */
    int &Array::put(int i, int j)
    {
        if(i<0 || i>=isize || j<0 || j>=jsize)
        {
            cout << "Ошибка, нарушены границы
массива!!!\n";
            exit(1);
        }
        return p[i * jsize + j];// возврат ссылки на p[
i ] [ j ]
    }
    // получение значения из массива
    int Array::get(int i, int j)
    {
        if(i<0 || i>=isize || j<0 || j>=jsize)
        {
            cout << "Ошибка, нарушены границы
массива!!!\n";
            exit(1);
        }
        return p[i * jsize + j]; // возврат символа
    }
    int main()
    {
        Array a(2, 3);
        int i, j;
        for(i=0; i<2; i++)
            for(j=0; j<3; j++)
                a.put(i, j) = i + j;
        for(i=0; i<2; i++)
```

```
        for(j=0; j<3; j++)
            cout << a.get(i, j) << ' ';
    // генерация ошибки нарушения границ массива
    a.put(10, 10);
    return 0;
}
```

Практическое задание №4

Цель работы: Научиться работать с массивами, указателями ссылками.

Постановка задачи: Создать статический или динамический массив, содержащий объекты классов согласно выбранному варианту задания. Реализовать функции ввода и отображения объектов массива.

Варианты заданий

№ варианта	Задание
1	Создать массив фиксированного размера, содержащий объекты типа «Дата». Класс «Дата» должен содержать несколько конструкторов. Инициализировать массив с помощью списка инициализации.
2	Создать массив фиксированного размера, содержащий объекты типа «Время». Класс «Время» должен содержать несколько конструкторов. Инициализировать массив с помощью списка инициализации.
3	Создать динамический массив, содержащий объекты типа «Автомобиль». Класс «Автомобиль» должен содержать несколько конструкторов. Реализовать функцию сортировки массива по какому-либо признаку.
4	Создать динамический массив, содержащий объекты типа «Фигура». Класс «Фигура» должен содержать несколько конструкторов. Реализовать функцию сортировки массива по какому-либо признаку.
5	Создать массив фиксированного размера, содержащий объекты типа «Фигура» и производных от него классов «Точка», «Квадрат» и «Круг». В базовом классе определить методы для отображения и перемещения фигуры. Разработать функцию для отображения и перемещения всех фигур массива.
6	Создать класс, содержащий массив объектов типа «Дата». Реализовать методы доступа к элементам массива с использованием ссылок.
7	Создать класс, содержащий массив объектов типа «Время». Реализовать методы доступа к элементам массива с использованием ссылок.
8	Создать класс, содержащий указатель на динамический массив объектов типа «Фигура». Реализовать методы доступа к элементам массива с использованием ссылок.
9	Создать класс, содержащий указатель на динамический

	массив объектов типа «Транспорт». Реализовать методы доступа к элементам массива с использованием ссылок.
0	Создать класс, содержащий указатель на динамический массив объектов типа «Фигура» и производными от него классами «Точка», «Квадрат» и «Круг». Реализовать методы доступа к элементам массива с использованием ссылок.

Практическое занятие 5. Шаблоны и исключительные ситуации

Шаблоны функций

Шаблоны, которые также называют родовыми или параметризованными типами, позволяют конструировать семейства функций и классов. В отличие от механизма перегрузки, когда для каждого набора формальных параметров устанавливается своя функция, шаблон семейства функций определяется один раз, но при этом оно параметризуется. Параметризовать в шаблоне функции можно тип возвращаемого функцией значения и типы параметров, количество и порядок которых должны быть фиксированы. В определении шаблона употребляется служебное слово template. Для параметризации используется список формальных параметров шаблона, который заключается в угловые скобки <>. Каждый формальный параметр шаблона помечен служебным словом class, за которым следует имя параметра.

Шаблон семейства функций состоит из двух частей – заголовка шаблона и определения функции, в котором тип возвращаемого значения и типы параметров обозначаются именами параметров шаблона.

```
template <class Ttype> тип имя_функции(список аргументов)
{/*тело функции*/}
```

Здесь Ttype – фиктивный тип, или список типов, через запятую, который используется при объявлении аргументов, локальных переменных и возвращаемых значений функции. Компилятор заменит этот фиктивный тип на один из реальных и создаст соответственно несколько перегружаемых функций, которые являются ограниченными, поскольку выполняют одни и те же действия. Например:

```
#include <iostream.h>
#include <string.h>
template <class X> int find(X object, X *list, int size)
{
	int i;
	for(i=0; i<size; i++)
		if(object == list[i]) return i;
	return -1;
}
int main()
{
	int a[ ]={1, 2, 3, 4};
	char *c="это проверка";
```

```
        double d[ ]={1.1, 2.2, 3.3};
        cout << find(3, a, 4) << endl;
        cout << find('a', c, strlen(c))<< endl;
        cout << find(0.0, d, 3);
        return 0;
}
```

Компилятор автоматически создает три перегруженные функции find(), соответствующие типам передаваемых аргументов.

Приведем пример определения шаблона функций, вычисляющих абсолютные значения числовых величин разных типов:

```
template <class type> type abs (type x) { return x > 0 ?
x: -x;}
```

В качестве еще одного примера рассмотрим шаблон семейства функций для обмена значений двух передаваемых им параметров.

```
template <class T> void swap (T* x, T* y)
{
        T z = *x;
        *x = *y;
        *y = x;
}
```

Здесь параметр T шаблона функций используется не только для спецификации формальных параметров, но и в теле определения функции, где он задает тип вспомогательной переменной z.

Если в программе присутствует приведенный ранее шаблон семейства функций swap() и появится последовательность операторов:

```
long k = 4, d = 8; swap (&k, &d);
```

то компилятор сформирует определение функции:

```
void swap (long* x, long* y)
{
        long z = *x;
        *x = *y;
        *y = z;
}
```

Затем будет выполнено обращение именно к этой функции, и значения переменных k и d поменяются местами.

Если в той же программе присутствуют операторы:

```
double a = 2.44, b = 66.3; swap (&a, &b);
```

то сформируется и выполнится функция

```
void swap (double* x, double* y){

      double x = *x;
      *x = *y;
      *y = x;
}
```

Параметры шаблона

Параметры шаблона являются формальными, а типы тех параметров, которые используются в конкретных обращениях к функции, служат фактическими параметрами шаблона. Имена параметров шаблона должны быть уникальными во всем его определении. В списке параметров шаблона функций их может быть несколько. Каждый из параметров должен начинаться со служебного слова class. Например, неверен заголовок:

```
template <class type1, type2, type3>
```

Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами, например:

```
template <class t, class t, class t>
```

Все параметры шаблона функций должны быть обязательно использованы в спецификациях параметров определения функции. Будет ошибочным шаблон:

```
template <class A, class B, class C> B func (A n, C m) {B
v; ... }
```

Применение параметра шаблона B в качестве типа возвращаемого функцией значения и для определения объекта v в теле функции недостаточно.

Шаблоны классов

Аналогично шаблонам функций определяется шаблон семейства классов:

```
template <список_параметров_шаблона> класс
```

В определении класса, входящего в шаблон, его имя является не именем отдельного класса, а параметризованным именем семейства классов.

С помощью шаблона класса можно создать класс, реализующий стек, очередь, дерево и т. д. для любых типов данных. Компилятор будет

генерировать правильный тип объекта на основе типа, задаваемого при создании объекта. Объявление шаблона класса имеет вид:

```
template <class Ttype> class имя_класса
{
     // поля и функции класса
}
```

Например:

```
// простой родовой связанный список
#include <iostream.h>
template <class data_t> class List
{
     data_t data;
     List *next;
public:
     List (data_t d);
     void add(List *node)
     {
         node->next = this;
         next=0;
     }
/* новый созданный элемент списка (объект) добавляет
к себе последний элемент, включенный в список */
     List *getnext()
     {
         return next;
     }
     data_t getdata()
     {
         return data;
     }
};
template <class data_t> List<data_t>::List(data_t d)
{
     data=d;
     next=0;
}
int main()
{
     //создается объект с реальным типом данных
     List<char> start('a');
     List <char> *p, *last;
     int i;
```

```
        // создание списка
        last=&start;
        for(i=1; i<26; i++)
        {
            p=new List<char> ('a'+i);
            p->add(last);
            last=p;
        }
        // вывод списка
        p=&start;
        while(p)
        {
            cout << p->getdata();
            p=p->getnext();
        }
        return 0;
    }
```

С помощью простого объявления можно создать другой объект, например, для хранения целых:

```
List <int> int start(1);
```

В список можно поместить структуры или другие объекты.

Класс-шаблон может иметь больше одного родового типа данных:

```
template<class T1,class T2> class M {T1 a; T2 b;}
```

В качестве аргумента в общем случае может быть использовано константное выражение, однако выражение, содержащее переменные, использовать в качестве фактического параметра шаблона нельзя.

Исключительные ситуации

Под исключительной ситуацией, или исключением (exception), понимают прерывание нормального потока программного управления в ответ на непредвиденное или аварийное событие. Исключение может порождаться ошибками, такими, как деление числа на нуль или обращением к памяти по недействительному адресу. В качестве ответа на ошибку функция, в которой возникла ошибка, инициирует (возбуждает) исключение оператором throw, за которым следует значение. Это значение может быть константой, переменной или объектом и предназначено для передачи информации обработчику исключения об исключении.

Обработчик исключения начинается ключевым словом catch с объявлением в круглых скобках. Если тип, определенный в этом объявлении,

совпадает с типом значения, данного в операторе throw, управление будет передано в блок, следующий за ключевым словом catch. В случае несовпадения типов, программа осуществляет поиск другого обработчика.

Для процедуры обработки исключения должен быть предусмотрен try-блок, в котором и происходит собственно инициализация исключения. Если исключение не инициализировано в try-блоке, программное управление проигнорирует catch-блок и непосредственно перейдет к первому оператору, находящемуся за catch-блоком.

```
try
{
     /*блок try*/
} catch(type1 arg){/*блок catch*/}
//операторы
```

С блоком try может связываться несколько блоков catch. Выполняется тот блок catch, для которого тип данных соответствует типу возникшей исключительной ситуации. При этом ее значение присваивается аргументу в круглых скобках блока catch. Если ошибка имеет место внутри блока try, она может генерироваться с помощью throw.

```
#include <iostream.h>
int main()
{
     try
     {
          // начало блока try
          cout << "Внутри блока try\n";
          throw 10;    // генерация ошибки 10
          cout << "Это выполнено не будет\n";
     }
     catch (int i)
     {
          // перехват ошибки
          cout << "Перехвачена ошибка номер: "<< i <<
"\n";
     }
     catch (char *s)
     {
          // перехват ошибки
          cout << "Перехвачена ошибка номер: "<<s <<
"\n";
     }
     return 0;
}
```

Будет выведено:

```
Внутри блока try
Перехвачена ошибка: 10
```

Оператор throw генерирует ошибку, после чего управление передано блоку catch. Если заменить тип ошибки int на double, ошибка перехвачена не будет. Для перехвата всех исключительных ситуаций независимо от типа можно воспользоваться многоточием, которое соответствует любому типу данных.

```
catch(…) {/*тело*/}
```

Ключевое слово throw после заголовка функции используется для определения списка инициализированных исключений:

```
void func( ) throw (except1, except2, except3, char*)
{       }
```

Функции, генерирующие исключения, вызываются из блока try. Описанная без оператора throw функция не может корректно для течения программы инициализировать исключение, а та, которая может это сделать, способна вызвать также исключение типа, производного от него. Блок catch, который перехватывает исключение для объектов, может перехватывать и исключения производных типов.

Если в функции инициализируется исключение, тип которого не совпадает с типом ни одного обработчика, то вызывается функция unexpected(), которая в свою очередь, вызывает terminate(), а terminate() – функцию abort() для завершения программы.

Практическое задание №5

Цель работы: Научиться работать с шаблонами классов и функций. Отлавливать и обрабатывать исключительные ситуации.

Постановка задачи: Написать программу с использованием шаблонов или обработчиков исключительных ситуаций согласно выбранному варианту задания.

Варианты заданий

№ варианта	Задание
1	Создать шаблон класса «Однонаправленный список». Реализовать списки объектов класса «Дата».
2	Создать шаблон класса «Стек». Реализовать стек из объектов класса «Время».
3	Создать шаблон класса «Массив» с параметром – количество элементов массива. Реализовать массив из объектов класса «Транспорт».
4	Создать шаблон класса «Двумерный массив» с параметрами – размерность массива. Реализовать массив из объектов класса «Фигура».
5	Создать шаблон функции для сортировки элементов массива. Выполнить сортировку массива объектов класса «Дата»
6	Создать шаблон функции для сортировки элементов массива. Выполнить сортировку массива объектов класса «Время»
7	Создать шаблон функции для сортировки элементов динамического массива. В качестве параметра шаблона указать размер массива .Выполнить сортировку массива объектов класса «Транспорт».
8	Создать класс «Время», содержащий метод для ввода времени с клавиатуры. При вводе некорректного времени, должна генерироваться исключительная ситуация. Реализовать обработчик исключительной ситуации.
9	Создать класс «Дата», содержащий метод для ввода даты с клавиатуры. При вводе некорректной даты, должна генерироваться исключительная ситуация. Реализовать обработчик исключительной ситуации.
0	Создать класс «Массив» для хранения объектов класса

	«Транспорт». Методы доступа к элементам массива должны генерировать исключение в случае некорректного индекса элемента массива. Реализовать обработчик исключительной ситуации.

Практическое занятие 6. Потоки и классы ввода/вывода

Стандартные потоки

В С++ ввод и вывод осуществляется через потоки – объекты классов ввода/вывода, которые передают и принимают данные и связываются с физическими устройствами. Следующие потоки определены и автоматически открываются при запуске приложения:

```
extern istream cin;   // стандартный поток ввода с клавиатуры
extern ostream cout;  // стандартный поток вывода на экран
extern ostream cerr;  //стандартный поток вывода сообщений
                      //об ошибках
extern ostream clog;//буферизованный поток вывода сообщений
                      //об ошибках */
```

Рассмотрим пример:

```
#include <iostream.h>
int main()
{
     int a,b,c;
     cout << "Hello, world\n";
     cin>>a>>b>>c;
     cout << "a/b+c=" << a*b-c << "\n";
     cout << "a^b|c=" << (a^b|c) << "\n";
     return 0;
}
```

Операция << пишет аргумент, строку "Hello, world" в стандартный поток вывода cout. Ввод производится с помощью операции >> стандартного потока ввода cin. Приоритет операции вставки в поток << достаточно низок, чтобы не использовать скобки для арифметических выражений. Для операций с более низкими приоритетами без скобок не обойтись.

Форматирование

Операция << применяется для неформатированного вывода данных стандартных типов. Операция определяет тип данных и выбирает подходящий формат. То же происходит и с операцией извлечения из потока >>. Помимо

этого существует несколько функций, преобразующих параметр в строку, которая используется для вывода.

```
char* oct(long, int=0);    //                  восьмеричное
представление
char* dec(long, int=0);    // десятичное представление
char* hex(long, int=0);    //             шестнадцатеричное
представление
char* chr(int, int=0);     // символ
char* str(char*, int=0);   // строка
```

Второй (необязательный) параметр указывает, сколько символьных позиций должно использоваться. Если задано поле нулевой длины, то для вывода потребуется столько позиций, сколько нужно, иначе будет производиться усечение или дополнение. Например:

```
cout <<"dec("<< x << ")=oct("<< oct(x,6)<<")= hex("
     << hex(x,4)<< ")";
```

Если x==15, то в результате получится:

```
dec(15) = oct( 17) = hex( f);
```

Для преобразования при выводе можно использовать строку в формате:

```
char* form(char* format, список параметров);
```

Аналогично стандартной функции printf() вывода языка C функция form() возвращает строку, получаемую в результате форматирования параметров, стоящих после первого управляющего параметра – строки формата format. Строка формата состоит из обычных символов, которые просто копируются в поток вывода, и спецификаций преобразования, влекущих преобразование и вывод параметра. Каждая спецификация преобразования начинается с символа %. Например:

```
cout<<form(" x= %d ",x);
```

Манипуляторы

Манипуляторы – функции потока, которые можно включать в операции помещения и извлечения в потоки (<<, >>), они бывают следующие:

```
endl //  помещение  в  выходной  поток  символа  конца
строки '\n'
ends // помещение в выходной поток символа '\0'
flush // вызов функции вывода буферизованных данных
```

```
dec, hex, oct // установка основания системы счисления
ws            // игнорирование при вводе пробелов
setbase(int)  // установка основания системы счисления
resetiosflasg(long)   // сброс флагов форматирования по маске
setiosflags(long) //установка флагов форматирования по маске
setfill(int)      // установка заполняющего символа
setprecision(int) //установка точности вывода
                  //вещественных чисел
setw(int)         // установка ширины поля ввода-вывода
```

Пример вызова манипулятора:

```
cout << 15 << hex << 15 << setbase(8) << 15;
cout<<hex<<100<<oct<<100<<dec<<100;
```

Ошибки потоков

Каждый поток (istream или ostream) имеет ассоциированное с ним состояние, с помощью проверки которого осуществляется обработка ошибок и нестандартных условий.

Поток может находиться в одном из следующих состояний:

```
enum stream_state { _good, _eof, _fail, _bad };
```

Если состояние _good или _eof, последняя операция ввода прошла успешно, а если – _good, то следующая операция ввода может пройти успешно, в противном случае она закончится неудачей. Другими словами, применение операции ввода к потоку, который не находится в состоянии _good, является пустой операцией. Состояние потока можно проверять, например, так:

```
switch (cin.rdstate())
{
  case _good: // последняя операция над cin прошла успешно
        break;
  case _eof: // конец файла
        break;
  case _fail: // некоего рода ошибка форматирования
        break;
  case _bad: // возможно, символы cin потеряны
        break;
```

```
}
```

Файловый ввод-вывод с применением потоков С++

В С++ существуют классы потоков ввода-вывода, определенные в соответствующей библиотеке:

```
ios           //базовый потоковый класс
streambuf     //буферизация потоков
istream       //потоки ввода
ostream       //потоки вывода
iostream      //двунаправленные потоки
istrstream        //строковые потоки ввода
ostrstream        //строковые потоки вывода
strstream     //двунаправленные строковые потоки
ifstream      //файловые потоки ввода
ofstream      //файловые потоки вывода
fstream       //двунаправленные файловые потоки
```

Стандартные потоки (istream,ostream,iostream) служат для работы с терминалом. Строковые потоки (istrstream, ostrstream, strstream) – для ввода-вывода из строковых буферов, размещенных в памяти, файловые потоки (ifstream, ofstream, fstream) – для работы с файлами.

Для реализации файлового ввода-вывода небходимо включить заголовочный файл fstream.h, содержащий производные от istream и ostream классы ifstream, ofstream и fstream, и объявить соответствующие объекты. Например:

```
ifstream in;  //ввод
ofstream out; //вывод
fstream io;   //ввод - вывод
```

Файловые потоки можно определить с помощью конструкторов:

```
ofstream obj (filename, mode),
ifstream obj (filename, mode),
```

где mode может иметь следующие значения:

```
ios::app      //запись в конец существующего файла
ios::ate      //после открытия файла перейти в его
конец
ios::binary   //открыть файл в двоичном режиме
              //(по умолчанию – текстовый)/
ios::in       //открыть для чтения
```

```
ios::nocreate //сообщать о невозможности открытия,
              //если файл не существует
ios::noreplace    //сообщать о невозможности открытия,
              //если файл существует
ios::out      //открыть для вывода
ios::trunc        //если файл существует, стереть
содержимое
```

При необходимости изменения способа открытия или применения файла можно при создании файлового потока использовать два или более флагов: ios::app|ios::noreplace

Для открытия файла одновременно для чтения и записи используются объекты класса fstream:

```
fstream obj(filename, ios::in|ios::app);
```

После объявления потоков открытие файла, связывающее его с потоком, можно производить не через конструктор, а с помощью метода open():

```
void open (char *filename,int mode,int access)
```

где filename – имя файла, включающее путь; mode – режим открытия файла, access – доступ. Параметр access принимает следующие значения: 0 – файл со свободным доступом, 1 –только для чтения, 8 – архивный файл.

Например:

```
ofstream out;
out.open("test.dat",ios::out,0);
```

Параметры можно задать по умолчанию.

```
out.open("test.dat");//будет то же самое
```

При завершении программы открытые файлы неявно закрываются. Для явного закрытия объектов файловых потоков применяется метод close().

Ввод-вывод в файлы

Для чтения-записи в потоки можно использовать перегружаемые операторы-функции >> и <<. Например:

```
#include <fstream.h>
int main()
{
```

```
/* создание  файла вывода с помощью конструктора */
    ofstream fout("test");
    if(!fout)
    {
        cout << "Файл открыть невозможно\n";
        return 1;
    }
    fout << "Привет!\n";
    fout << 100 << ' ' << hex << 100 << endl;
    fout.close();
    ifstream fin("test"); // открытие обычного файла
ввода
    if(!fin)
    {
        cout << "Файл открыть невозможно\n";
        return 1;
    }
    char str[80];
    int i;
    fin >> str >> i;
    cout << str << ' ' << i << endl;
    fin.close();
    return 0;
}
```

Сам оператор << можно перегрузить, как в следующем примере:

```
//создание   недружественной   функции-вставки   для
объектов Coord
include <iostream.h>
class Coord
{
    public:
    int x,y;//должны быть открытыми
    Coord()
    {
        x=0;
        y=0;
    }
    Coord(int i,int j)
    {
        x=i;
        y=j;
    }
};
```

```
//функция вставки для объектов класса coord
ostream &operator<<(ostream &stream, Coord ob)
{
      stream <<ob.x<<”,”<<ob.y<<’\n’;
      return stream;
}
int main()
{
      Coord a(1,1), b(10,23);
      cout <<a<<b;
      return 0;
}
```

Оператор ввода (извлечения) также можно перегрузить:

```
istream &operator>>(istream &stream, имя_класса ob)
{
      //тело функций ввода
      return stream;
}
```

При работе с файлами возможно использование методов ввода-вывода одного символа: istream &get(char &ch); и ostream &put(char ch);

Пример использования этих методов и аргументов командной строки:

```
#include <iostream.h>
#include <fstream.h>
int main(int argc, char *argv[ ])
{
      char ch;
      if(argc!=2)
      {
          cout << "Использование:WRITE<имя_файла>\n";
          return 1;
      }
      ofstream out(argv[1]);
      if(!out)
      {
          cout << "Файл открыть невозможно\n";
          return 1;
      }
      cout << "Для остановки введите символ $\n";
      do
      {
          cout << ": ";
```

```
            cin.get(ch);
            out.put(ch);
        } while (ch!='$');
        out.close();
        return 0;
    }
```

Для записи-считывания блоков двоичных данных используются функции, которые считывают-записывают n байт в буфер или из буфера:

```
    istream &read(unsigned char *buf, int n);
    ostream &write(const unsigned char *buf, int n);

    #include <iostream.h>
    #include <fstream.h>
    #include <string.h>
    int main()
    {
        //   не   будет   нежелательных   преобразований
символов
        // при вводе и выводе
        ofstream out("test", ios::binary);
        if(!out)
        {
            cout << "Файл открыть невозможно\n";
            return 1;
        }
        double num = 100.45;
        char str[ ] = "Это проверка";
        out.write((char *) &num, sizeof(double));
        out.write(str, strlen(str));
        out.close();
        return 0;
    }
```

Для позиционирования в файле имеются методы seekg(), seekp(), tellp(). При этом seekg() назначает или возвращает текущую позицию указателя чтения, а seekp() – текущую позицию указателя записи. Обе функции могут иметь один или два аргумента. При вызове с одним аргументом функции перемещают указатель на заданное место, а при вызове с двумя аргументами вычисляется относительная позиция от начала файла (ios::beg) текущей позиции (ios::cur) или от конца файла (ios::end). Текущая позиция определяется методом tellp().

Для объектов файловых потоков контроль состояния производится с помощью методов, манипулирующих флагами ошибок:

`bad()` – возвращает ненулевое значение, если обнаружена ошибка;
`clear()` – сбрасывает сообщения об ошибках;
`eof()` – возвращает ненулевое значение, если обнаружен конец файла;
`fail()` – возвращает ненулевое значение, если операция завершилась неудачно;
`good()` – возвращает ненулевое значение, если флаги ошибок не выставлены;
`rdstate()` – возвращает текущее состояние флагов ошибок.

Если флаги показывают наличие ошибки, все попытки поместить в поток новые объекты будут игнорироваться.

Практическое задание №6

Цель работы: Научиться использовать потоки и классы ввода-вывода стандартной библиотеки языка C++.

Постановка задачи: Написать программу с использованием потоков и классов согласно выбранному варианту задания.

Варианты заданий

№ варианта	Задание
1	Создать класс «Массив» для хранения строк. Реализовать методы для вывода массива строк в текстовом виде и в виде шестнадцатеричного дампа.
2	Создать класс «Стек» для хранения строк. Реализовать методы для вывода содержимого стека строк в текстовом виде и в виде шестнадцатеричного дампа.
3	Создать класс «Однонаправленный список» для хранения строк. Реализовать методы для вывода массива строк в текстовом виде и в виде последовательности десятичных кодов символов.
4	Создать класс «Стек» для хранения комплексных чисел. Реализовать методы чтения содержимого стека из файла и записи содержимого стека в файл.
5	Создать класс «Массив» для хранения векторов. Реализовать методы чтения массива векторов из файла и записи массива векторов в файл.
6	Создать класс «Однонаправленный список» для хранения объектов класса «Время». Реализовать методы чтения объектов списка из файла и записи объектов списка в файл.
7	Создать класс «Стек» для хранения объектов класса «Дата». Реализовать методы чтения содержимого стека из файла и записи содержимого стека в файл.
8	Создать класс «Массив» для хранения объектов класса «Транспорт». Реализовать методы чтения объектов массива из файла и записи объектов массива в файл.
9	Создать массив фиксированного размера для хранения объектов класса «Фигура» и производных от него: «Точка», «Квадрат», «Круг». Написать функцию для записи массива объектов в файл и чтения массива

	объектов их файла.
0	Создать динамический массив для хранения объектов класса «Транспорт» и производных от него: «Автомобиль», «Самолет», «Поезд». Написать функцию для записи массива объектов в файл и чтения массива объектов их файла.

Практическое занятие 7. Статические и константные члены, локальные классы

Статические члены

Класс – это тип, а не объект данных, и в каждом объекте класса имеется своя собственная копия данных, членов этого класса. Иногда необходимо, чтобы некоторые данные-члены класса были одинаковыми (разделяемыми) для всех экземпляров класса. Предпочтительно, чтобы такие разделяемые данные были описаны как часть класса. Возможно применение статических методов класса, которые можно вызывать не через объект (даже тогда, когда объект существует), а через имя класса. Для объявления членов класса статическим применяется модификатор static. Статические данные-члены класса должны быть объявлены в классе и определены за пределами класса (внутренние определения запрещены). Обычные спецификаторы доступа действуют и для статических членов класса. Память, занимаемая статическими данными класса, не учитывается при определении размера объекта с помощью операции sizeof.

```
class My
{
	static int count;
public:
	static int getCount(){return count;}
};
int My::count = 0;
```

Так как статические элементы-функции не ассоциируются с отдельными представителями, то при вызове им не передается указатель this. Из этого следует, что:

1. статическая функция член класса может обращаться только к статическим данным класса и вызывать только другие статические функции текущего класса;
2. статическая функция не может быть объявлена как virtual или const.

Использование статических членов класса может заметно снизить потребность в глобальных переменных, применение которых нежелательно в принципе.

Константные члены и объекты

Можно создавать объекты класса с модификатором const, информирующим компилятор о том, что содержимое объекта не должно изменяться после инициализации. Чтобы предотвратить изменение значений элементов константного объекта, компилятор сообщает об ошибке, если объект

используется с неконстантной функцией-элементом. Возможно использование константных указателей и указателей на константный объект.

```
    My  *p01;//  константный  указатель,  который  нельзя
изменять
    p01= new My;//константный объект
    My* const p03=p01;
    const My ob;
    const My *p04=&ob;
    /*  указатель  на  константный  объект,  т.е.  объект
нельзя  изменить,  используя  данный  указатель,  но  можно
изменять сам указатель p04*/
    const My* const p01=&ob;
    /* указатель-константа на константный объект: нельзя
изменять и вызывать неконстантные функции*/
```

Константная функция-элемент в свою очередь:

1. объявляется с ключевым словом const, которое следует за списком параметров;
2. не может изменять значение полей класса;
3. не может вызывать неконстантные функции-члены класса;
4. может вызываться как для константных, так и неконстантных объектов класса.

Следующий пример демонстрирует константные функции и объекты:

```
class Point1
{
     int x;
public:
     Point1(int x){Point1::x=x;}
     void setX(int x){Point1::x=x;}
     void getX(int &x1)const;
};
void Point1::getX(int &x1) const
{
     x1=x;
}
int main(int argc, char* argv[])
{
     Point1 ob1(5);
     const Point1 ob2(10);
     int x2;
     ob1.getX(x2);
     //ob2.setX(x2);//ошибка:   вызов   неконстантной
функции
```

```
                    // неконстантным объектом
        ob2.getX(x2);
        return 0;
}
```

Однако следует отметить, что константный метод может изменять данные-члены класса, которые описаны с модификатором mutable, предшествующим типу данных:

```
mutable int i;
```

Локальные классы

Если класс объявляется вне любого блока, то он называется глобальным. В то же время класс может объявляться внутри функции, называясь при этом внутренним. Область видимости внутреннего класса ограничивается функцией, в которой он определен. Если класс объявляется внутри другого класса, то он является вложенным. Его область видимости – класс, в котором он определен. Внутренние и вложенные классы называют локальными.

Пример описания вложенного класса:

```
class Student
{
        int id;
        class Exam
        {
            int idExam;
            char name[80];
        public:
            Exam (int idExam, char* s)
            {
                Exam::idExam=idExam;//определение
видимости
                strcpy(name, s);
            }
        };
        Exam* first;
public:
        void add(int n, char *s)
        {
            first = new Exam(n, s);
        }
};
```

Внутри локальных классов можно использовать типы, статические и внешние (extern) переменные, внешние функции и элементы перечислений из области, где локальный класс описан. В то же время нельзя использовать автоматические переменные из указанной области. В локальных классах можно определить статические методы. Во вложенных классах можно также использовать статические переменные. Внутренний класс не может иметь статических переменных. Методы локального класса определяются только внутри класса.

Если один класс вложен в другой класс, то это не дает каких-либо особых прав доступа к элементам друг друга. Обращение может выполняться по общим правилам.

```
class OuterClass
{
     class InnerClass
     {
          //во вложенном классе можно использовать
          //статические переменные
          static double d;
     };//конец объявления вложенного класса
}; // конец объявления внешнего класса
     /*определение статической переменной */
     double OuterClass::InnerClass::d=5.32;
     void extf()
     {
          //Внешняя функция
          class InnerClass1
          {
          static double d1;    //ошибка:        нельзя
определить
                               //переменную      d1      за
пределами
                                    //функции extf();
          }; //конец объявления внутреннего класса
};
int main()
{
     OuterClass oc;
     extf();
}
```

Если только вложенный класс не является очень простым, то в таком описании трудно разобраться. Рекомендуется не использовать сложные описания локальных классов и не использовать в иерархии вложенности более одной ступени. Кроме того, вложение классов – это не более чем соглашение о

записи, поскольку вложенный класс не является скрытым в области видимости лексически охватывающего класса.

```
class Student
{
      int id;
      class Exam
      {
           int idExam;
           char name[80];
      public:
           Exam (int idExam, char* s);
      };
      Exam* first;
public:
      void add(int n, char *s)
      {
           first = new Exam(n, s);
      }
};
Student::Exam::Exam (int idExam, char* s)
{
      Exam::idExam=idExam;
      strcpy(name, s);
}
```

Большую часть нетривиальных классов лучше описывать раздельно, и при этом существует возможность обеспечить доступ к закрытым данным-членам некоторого класса сразу для всех методов другого класса. Такой класс объявляется дружественным для первого класса.

```
class Exam
{
      friend class Student;//объявление дружественного 
класса
      int idExam;
      char name[80];
public:
      Exam (int idExam, char* s)
      {
           Exam::idExam=idExam;
           strcpy(name, s); }
      };
class Student
{
```

```
    int id;
    Exam* first;
public:
    void add(int n, char *s)
    {
        first = new Exam(n, s);
    }
};
```

Практическое задание №7

Цель работы: Научиться работать со статическими и константными членами классов. Научиться объявлять локальные классы.

Постановка задачи: Написать программу с использованием статических и константных членов класса, а также локальных классов. Создать класс согласно выбранному варианту задания. Реализовать методы чтения и отображения данных.

Варианты заданий

№ варианта	Задание
1	Создать класс «Однонаправленный список» для хранения объектов класса «Время». Класс «Время» должен содержать статический член данных для подсчета количества экземпляров объектов данного класса.
2	Создать класс «Стек» для хранения объектов класса «Дата». Класс «Дата» должен содержать статический член данных для подсчета количества экземпляров объектов данного класса.
3	Создать класс «Стек» для хранения объектов класса «Транспорт». Класс «Транспорт» должен содержать статический член данных для подсчета количества объектов данного класса.
4	Создать класс «Время», содержащий только один приватный конструктор. Реализовать статический метод для создания объектов класса.
5	Создать класс «Дата», содержащий только один приватный конструктор. Реализовать статический метод для создания объектов класса.
6	Создать класс «Транспорт», содержащий только один приватный конструктор. Реализовать статический метод для создания объектов класса.
7	Создать класс «Дата и Время», содержащий вложенные классы «Дата» и «Время».
8	Создать класс «Транспортные средства», содержащий вложенные классы «Автомобиль», «Самолет», «Поезд».

9	Создать класс «Фигуры», содержащий вложенные классы «Точка», «Квадрат», «Круг».
0	Создать класс «Компьютер», содержащие вложенные классы, описывающие компоненты компьютера.

Практическое занятие №8. Работа с файлами.

Цель: овладеть возможностью считывания/записи данных из/в файл.

Ход работы:

1. Рассмотреть описанные в теоретических сведениях примеры.
2. Реализовать возможность записи/чтения экземпляра класса, созданного самостоятельно.

Теоретические сведения.

Потоки: байтовые, символьные, двоичные

Большинство устройств, предназначенных для выполнения операций ввода–вывода, являются байт–ориентированными. Этим и объясняется тот факт, что на самом низком уровне все операции ввода–вывода манипулируют с байтами в рамках **байтовых потоков**.

С другой стороны, значительный объём работ, для которых, собственно и используется вычислительная техника, предполагает работу с символами, а не с байтами (заполнение экранной формы, вывод информации в наглядном и легко читаемом виде, текстовые редакторы).

Символьно–ориентированные потоки, предназначенные для манипулирования с символами, а не с байтами, являются потоками ввода–вывода более высокого уровня. В рамках Framework.NET определены соответствующие классы, которые при реализации операций ввода–вывода обеспечивают автоматическое преобразование данных типа byte в данные типа char и обратно.

В дополнение к байтовым и символьным потокам в C# определены два класса, реализующих механизмы считывания и записи информации непосредственно в двоичном представлении (потоки BynaryReader и BynaryWriter).

Общая характеристика классов потоков

Основные особенности и правила работы с устройствами ввода–вывода в современных языках высокого уровня описываются в рамках классов потоков. Для языков платформы .NET местом описания самых общих свойств потоков является **класс System.IO.Stream**.

Назначение этого класса заключается в объявлении общего стандартного набора операций (стандартного интерфейса), обеспечивающих работу с устройствами ввода–вывода, независимо от их конкретной реализации источников и получателей информации.

В рамках Framework.NET, независимо от характеристик того или иного устройства ввода–вывода, **программист ВСЕГДА может узнать**:

можно ли читать из потока – bool CanRead (если можно – значение должно быть установлено в true),

можно ли писать в поток – bool CanWrite (если можно – значение должно быть установлено в true),

можно ли задать в потоке текущую позицию – bool CanSeek (если последовательность, в которой производится чтение–запись не является жёстко детерминированной и возможно позиционирование в потоке – значение должно быть установлено в true),

позицию текущего элемента потока – long Position (возможность позиционирования в потоке предполагает возможность программного изменения значения этого свойства),

общее количество символов потока (длину потока) – long Length.

В соответствии с общими принципами реализации операций ввода–вывода, **для потока** предусмотрен **набор методов, позволяющих реализовать**:

Метод потока	Назначение
int ReadByte()	чтение байта из потока с возвращением целочисленного представления СЛЕДУЮЩЕГО ДОСТУПНОГО байта в потоке ввода
int Read(byte[] buff, int index, int count)	чтение определённого (count) количества байтов из потока и размещение их в массиве buff, начиная с элемента buff[index], с возвращением количества успешно прочитанных байтов
Write(byte[] buff, int index, int count)	запись в поток одного байта
int void WriteByte(byte b)	запись в поток определённого (count) количества байтов из массива buff, начиная с элемента buff[index], с возвращением количества успешно записанных байтов
long Seek (long index, SeekOrigin origin)	позиционирование в потоке **–**(позиция текущего байта в потоке задаётся значением смещения index относительно позиции origin)
void Flush()	для буферизованных потоков принципиальна операция флэширования (записи содержимого буфера потока на физическое устройство)
void Close()	закрытие потока

Множество классов потоков ввода–вывода в Framework.NET основывается (наследует свойства и интерфейсы) на **абстрактном классе Stream**. При этом классы конкретных потоков обеспечивают собственную реализацию интерфейсов этого абстрактного класса.

Наследниками класса Stream являются, в частности, три класса байтовых потоков:

BufferedStream – обеспечивает буферизацию байтового потока. Как правило, буферизованные потоки являются более производительными по сравнению с небуферизованными,

FileStream – байтовый поток, обеспечивающий файловые операции ввода–вывода,

MemoryStream – байтовый поток, использующий в качестве источника и хранилища информации оперативную память.

Манипуляции с потоками предполагают НАПРАВЛЕННОСТЬ производимых действий. Информацию из потока можно ПРОЧИТАТЬ, а можно её в поток ЗАПИСАТЬ. Как чтение, так и запись, предполагают реализацию определённых механизмов байтового обмена с устройствами ввода–вывода.

Свойства и методы, объявляемые в соответствующих классах, **определяют специфику потоков**, используемых для чтения и записи:

TextReader,

TextWriter.

Эти классы являются абстрактными. Это означает, что они не "привязаны" ни к какому конкретному потоку. Они лишь определяют интерфейс (набор методов), который позволяет организовать чтение и запись информации для любого потока.

Класс Filestream

В C# предусмотрены классы, которые позволяют считывать содержимое файлов и записывать в них информацию. Конечно же, дисковые файлы — самый распространенный тип файлов. На уровне операционной системы все файлы обрабатываются на побайтовой основе.

Чтобы создать байтовый поток с привязкой к файлу, используйте класс FileStream. Класс **FileStream** — производный от Stream и потому обладает функциональными возможностями базового класса. Помните, что потоковые классы, включая FileStream, определены в пространстве имен **System.IO**.

Следовательно, при их использовании в начало программы вы должны включить следующую инструкцию:

using System.IO;

Открытие и закрытие файла.

Для начала необходимо создать объект класса **FileStream**, выбрав наиболее подходящий конструктор. Самый распространенный:

```
FileStream(string filename, FileMode mode)
```

Например, `FileStream fin = new FileStream("test.dat", FileMode.Open);`

Значения FileMode enumeration, описывающие, каким образом операционная система должна открывать файл:

Имя члена	Описание
Append	Открывается существующий файл, и выполняется поиск конца файла, или же создается новый файл. FileMode.Append можно использовать только вместе с FileAccess.Write. Любая попытка чтения заканчивается неудачей и создает исключение

	ArgumentException.
Create	Описывает, что операционная система должна создавать новый файл. Если файл уже существует, он будет переписан. Это требует FileIOPermissionAccess.Write и FileIOPermissionAccess.Append. System.IO.FileMode.Create эквивалентно следующему запросу: если файл не существует, использовать CreateNew; в противном случае использовать Truncate.
CreateNew	Описывает, что операционная система должна создать новый файл. Это требует FileIOPermissionAccess.Write. Если файл уже существует, создается исключение IOException.
Open	Описывает, что операционная система должна открыть существующий файл. Возможность открыть данный файл зависит от значения, задаваемого FileAccess. Если данный файл не существует, создается исключение System.IO.FileNotFoundException.
OpenOrCreate	Указывает, что операционная система должна открыть файл, если он существует, в противном случае должен быть создан новый файл. Если файл открыт с помощью FileAccess.Read, требуется FileIOPermissionAccess.Read. Если файл имеет доступ FileAccess.ReadWrite и данный файл существует, требуется FileIOPermissionAccess.Write. Если файл имеет доступ FileAccess.ReadWrite и файл не существует, в дополнение к Read и Write требуется FileIOPermissionAccess.Append .
Truncate	Описывает, что операционная система должна открыть существующий файл. После открытия должно быть выполнено усечение файла таким образом, чтобы его размер стал равен нулю. Это требует FileIOPermissionAccess.Write. Попытка чтения из файла, открытого с помощью Truncate, вызывает исключение.

Если необходимо ограничить доступ только чтением или только записью, используйте следующий конструктор:

FileStream(string filename, FileMode mode,FileAccess how)

Где FileAccess enumerations определяет доступ к файлу:

Члены перечисления	Описание
Read	Read access to the file. Data can be read from the file. Combine with Write for read/write access.
ReadWrite	Read and write access to the file. Data can be written to and read from the file.
Write	Write access to the file. Data can be written to the file. Combine with Read for read/write access.

Например,

```
FileStream fin = new FileStream("test.dat",FileMode.Open,FileAccess.Read)
```

По завершении работы с файлом его необходимо закрыть. Для этого достаточно вызвать метод Close(). При закрытии файла освобождаются системные ресурсы, ранее выделенные для этого файла, что дает возможность использовать их для других файлов. Метод Close() может генерировать исключение типа lOException.

```
fin.Close();
```

В **классе FileStream** определены два метода, которые считывают/записывают байты из файла:

ReadByte() / WriteByte() - чтобы прочитать/записать из файла один байт

Read () / Write() - чтобы считать/записать блок байтов.

При выполнении операции вывода в файл выводимые данные зачастую не записываются немедленно на реальное физическое устройство, а буферизируются операционной системой до тех пор, пока не накопится порция данных достаточного размера, чтобы ее можно было всю сразу переписать на диск. Такой способ выполнения записи данных на диск повышает эффективность системы. Например, дисковые файлы организованы по секторам, которые могут иметь размер от 128 байт. Данные, предназначенные для вывода, обычно буферизируются до тех пор, пока не накопится такой их объем, который позволяет заполнить сразу весь сектор.

Но если вы хотите записать данные на физическое устройство вне зависимости от того, полон буфер или нет, вызовите следующий метод Flush ():

```
void Flush()
```

```
// Запись данных в файл.
   class WriteToFile
   {
      public static void Main(string[] args)
      {
         FileStream fout;
         // Открываем выходной файл
         try
         {
            fout = new FileStream("test.txt", FileMode.Create);
         }
         catch (IOException exc)
         {
            Console.WriteLine(exc.Message + "Ошибка при открытии выходного файла.");
            return;
         }
```

```
            // Записываем в файл алфавит,
            try
            {
               for (char c = 'A'; c <= 'Я'; c++) fout.WriteByte((byte)c);
            }
            catch (IOException exc)
            {
               Console.WriteLine(exc.Message + "Ошибка при записи в файл.");
               fout.Close();
            }

        }
    }
```

Эта программа сначала открывает для вывода файл с именем test.txt . Затем в этот файл записывается алфавит английского языка, после чего файл закрывается.

Обратите внимание на то, как обрабатываются возможные ошибки с помощью блоков try/catch.

Копирование файла.

Одно из достоинств байтового ввода-вывода с использованием класса FileStream заключается в том, что этот класс можно использовать для всех типов файлов, а не только текстовых.

```
public static void Main(string[] args)
    {
        int i;
        FileStream fin = new FileStream(args[0], FileMode.Open);
        FileStream fout = new FileStream(args[1], FileMode.Create);
        // Копируем файл,
        try
        {
            do
            {
                i = fin.ReadByte();
                if (i != -1) fout.WriteByte((byte)i);
            } while (i != -1);
        }
        catch (IOException exc)
        {
            Console.WriteLine(exc.Message + "Ошибка при чтении файла. ");
            fin.Close();
            fout.Close();
        }
```

Файловый ввод–вывод с ориентацией на символы.

Следующие методы определяют базовые механизмы символьного ввода–вывода.

Для класса TextReader:

Методы класса **TextReader**	Назначение
int Peek()	Reads the next character without changing the state of the reader or the character source. Returns the next available character without actually reading it from the input stream
int Read()	Перегруженные. Несколько одноименных функций с одним и тем же именем. Читает значения из входного потока. Вариант int Read() предполагает чтение из потока одного символа с возвращением его целочисленного эквивалента или –1 при достижении конца потока. Вариант int Read (char[] buff, int index, int count) и его полный аналог int ReadBlock(char[] buff, int index, int count) обеспечивает прочтение a maximum of count characters из текущего потока и записывает данные в buffer, начиная at index
string ReadLine()	Читает строку символов из текущего потока. Возвращается ссылка на объект типа string
string ReadToEnd()	Читает все символы, начиная с текущей позициисимвольного потока, определяемого объектом класса TextReader и возвращает результат как ссылка на объект типа string
void Close()	Закрывает поток ввода

Для класса TextWriter:

Методы класса **TextWriter**	Назначение
void Write()	множество перегруженных вариантов функции со значениями параметров, позволяющих записывать символьное представление значений базовых типов (смотреть список базовых типов) и массивов значений (в том числе и массивов строк)
void Flush()	Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream. Очистка буфера вывода с предварительным выводом в поток вывода (носитель данных) содержимого буфера
void Close()	Закрывает поток вывода.

Эти классы являются базовыми для классов:

StreamReader – содержит свойства и методы, обеспечивающие считывание СИМВОЛОВ из байтового потока,

StreamWriter – содержит свойства и методы, обеспечивающие запись СИМВОЛОВ в байтовый поток.

Интересно заметить, что у всех ранее перечисленных классов имеются методы, обеспечивающие закрытие потоков и не определены методы обеспечивающие открытие соответствующего потока. Потоки открываются в момент создания объекта–представителя соответствующего класса. Наличие функции, обеспечивающей явное закрытие потока принципиально. Оно связано с особенностями выполнения управляемых модулей в Framework.NET. Время начала работы сборщика мусора заранее неизвестно.

Пример использования StreamWriter.

Рассмотрим простую утилиту, которая считывает строки текста, вводимые с клавиатуры, и записывает их в файл test.txt. Текст считывается до тех пор, пока пользователь не введет слово "стоп". Здесь используется объект класса FileStream, помещенный в оболочку класса StreamWriter для вывода данных в файл.

```
public static void Main(string[] args)
    {
      string str;
      FileStream fout = new FileStream("test.txt", FileMode.Create);
      StreamWriter fstr_out = new StreamWriter(fout);
      Console.WriteLine("Введите текст 'стоп' для завершения).");
      do
      {
        Console.Write(": ");
        str = Console.ReadLine();
        if (str != "стоп")
        {
          str = str + "\r\n"; // Добавляем символ новой строки,
          try
          {
            fstr_out.Write(str);
          }
          catch (IOException exc)
          {
            Console.WriteLine(exc.Message + "Ошибка при работе с
файлом.");
            return;
          }
        }
      } while (str != "стоп");
       fstr_out.Close() ;
      }
```

Пример использования StreamReader.

Следующая программа считывает текстовый файл test.txt и отображает его содержимое на экране.

```
public static void Main(string[] args)
    {
       string s;
       FileStream fin = new FileStream("test.txt", FileMode.Open);
       StreamReader fstr_in = new StreamReader(fin);
       // Считываем файл построчно
       while ((s = fstr_in.ReadLine()) != null)
       {
          Console.WriteLine(s);
       }
       fstr_in.Close();

    }
```

Пример перенаправления потоков.

Такие стандартные потоки, как Console.In, можно перенаправлять. Безусловно, чаще всего они перенаправляются в какой-нибудь файл. При перенаправлении стандартного потока входные и/или выходные данные автоматически направляются в новый поток. При этом устройства, действующие по умолчанию, игнорируются. Благодаря перенаправлению стандартных потоков программа может считывать команды из дискового файла, создавать системные журналы или даже считывать входные данные с сетевых устройств.

Перенаправить стандартный поток можно двумя способами. Во-первых, при вы- полнении программы из командной строки можно использовать операторы "<" и ">".

В случае программного перенаправления потоков используются:

static void Setln(TextReader *input)*

static void SetOut(TextWriter *output)*

static void SetError(TextWriter *output)*

Пример, который перенаправляет выходной поток в файл:

```
// Перенаправление потока Console.Out.
       StreamWriter log_out;
       try
       {
          log_out = new StreamWriter("logfile.txt");
       }
       catch (IOException exc)
       {
          Console.WriteLine(exc.Message + "Не удается открыть файл.");
          return;
       }
       // Направляем стандартный выходной поток в системный журнал.
```

```
Console.SetOut(log_out);
Console.WriteLine("Это начало системного журнала.");
for(int i=0; i<10; i++) Console.WriteLine(i);
Console.WriteLine("Это конец системного журнала.");
    log_out.Close();
```

Содержимым файла logfile.txt будет:

```
Это начало системного журнала.
0
1
2
3
4
5
6
7
8
9
Это конец системного журнала.
```

И в файл можно записать информацию, используя привычные классы и методы!

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

namespace OutFile
{
class Program
{

static void Main(string[] args)
{
string buff;

FileStream outFstr, inFstr; // Ссылки на файловые потоки.

// Ссылка на выходной поток. Свойства и методы,
// которые обеспечивают запись в...
StreamWriter swr;

// Ссылка на входной поток. Свойства и методы,
// которые обеспечивают чтение из...
```

```
StreamReader sr;

// Класс Console - Средство управления ПРЕДОПРЕДЕЛЁННЫМ
потоком.
// Сохранили стандартный выходной поток,
// связанный с окошком консольного приложения.
TextWriter twrConsole = Console.Out;

// Сохранили стандартный входной поток, связанный с буфером
клавиатуры.
TextReader trConsole = Console.In;

inFstr = new FileStream
        (@"F:\Users\Work\readme.txt", FileMode.Open, FileAccess.Read);
sr = new StreamReader(inFstr); // Входной поток, связанный с файлом.

outFstr = new FileStream
        (@"F:\Users\Work\txt.txt", FileMode.Create, FileAccess.Write);
swr = new StreamWriter(outFstr); // Выходной поток, связанный с файлом.

// А вот мы перенастроили предопределённый входной поток.
// Он теперь связан не с буфером клавиатуры, а с файлом, открытым для
чтения.
Console.SetIn(sr);
Console.SetOut(swr);

while (true)
{
// Но поинтересоваться в предопределённом потоке отностительно
// конца файла невозможно.
// Такого для предопределённых потоков просто не предусмотрено.
if (sr.EndOfStream) break;

// А вот читать - можно.
buff = Console.ReadLine();
Console.WriteLine(buff);

}


Console.SetOut(twrConsole);
Console.WriteLine("12345");
Console.SetOut(swr);
Console.WriteLine("12345");
Console.SetOut(twrConsole);
```

```
Console.WriteLine("67890");
Console.SetOut(swr);
Console.WriteLine("67890");

sr.Close();
inFstr.Close();
swr.Close();
outFstr.Close();
}
}
}
```

ЛИТЕРАТУРА

Перечень основной литературы:

1. Иванова Г.С. Объектно-ориентированное программирование [Электронный ресурс]: учебник/ Иванова Г.С., Ничушкина Т.Н. — Электрон. текстовые данные. — Москва: Московский государственный технический университет имени Н.Э. Баумана, 2014. — 456 c. — Режим доступа: http://www.iprbookshop.ru/94030.html. — ЭБС «IPRbooks».
2. Маляров А.Н. Объектно-ориентированное программирование [Электронный ресурс]: учебник для технических вузов/ Маляров А.Н.— Электрон. текстовые данные. — Самара: Самарский государственный технический университет, ЭБС АСВ, 2017.— 332 c.— Режим доступа: http://www.iprbookshop.ru/91772.html.— ЭБС «IPRbooks».
3. Мурадханов С.Э. Информатика и программирование: объектно-ориентированное программирование (на основе языка C#) [Электронный ресурс]: учебник/ Мурадханов С.Э., Широков А.И. — Электрон. текстовые данные. — Москва: Издательский Дом МИСиС, 2015. — 309 c. — Режим доступа: http://www.iprbookshop.ru/98855.html.— ЭБС «IPRbooks».

Перечень дополнительной литературы:

1. Зыков С.В. Введение в теорию программирования. Объектно-ориентированный подход [Электронный ресурс]: учебное пособие/ Зыков С.В.— Электрон. текстовые данные.— Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021.— 187 c.— Режим доступа: http://www.iprbookshop.ru/102007.html.— ЭБС «IPRbooks».
2. Николаев Е.И. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие/ Николаев Е.И. — Электрон.текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2015. — 225 c. — Режим доступа: http://www.iprbookshop.ru/62967.html. — ЭБС «IPRbooks».
3. Сорокин А.А. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие. Курс лекций/ Сорокин А.А. — Электрон.текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2014. — 174 c. — Режим доступа: http://www.iprbookshop.ru/63110.html. — ЭБС «IPRbooks».

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к практическим занятиям
по дисциплине
«Объектно-ориентированное программирование»

для направления подготовки
09.03.02 Информационные системы и технологии
Направленность (профиль)
«Информационные системы и технологии в бизнесе»

Часть 2

Невинномысск, 2023

СОДЕРЖАНИЕ

стр

ВВЕДЕНИЕ 3

Практическое занятие №1 Создание главного окна приложения в среде C# 4

Практическое занятие №2 Создание главного меню приложения 19

Практическое занятие №3 Создание многооконного приложения 24

Практическое занятие №4 Создание пользовательских диалоговых окон 28

Практическое занятие №5 Создание панели инструментов и контекстного меню 39

Практическое занятие №6 Создание строки состояния 44

Практическое занятие №7 Создание элементов управления 49

Практическое занятие №8 Подготовка ADO.NET к работе в приложении 75

Список литературы 90

ВВЕДЕНИЕ

Работа на практических занятиях по учебной дисциплине «Объектно-ориентированное программирование» предполагает изучение студентами основ объектно-ориентированного программирования.

Изучение дисциплины предполагает формирование компетенции

Код, формулировка компетенции	Код, формулировка индикатора
ПК-4 Способен разработать архитектуру ИС	ИД-1$_{\text{ПК-4}}$ Осуществляет разработку стратегии развития информационных технологий инфраструктуры предприятия и управления ее реализацией
	ИД-2$_{\text{ПК-4}}$ Осуществляет разработку архитектуры ИТ и ИС инфраструктуры предприятия
	ИД-3$_{\text{ПК-4}}$ Осуществляет обоснование архитектуры ИС

Выполнение практических заданий включает:

1. Изучение студентами необходимого теоретического материала по теме лабораторной работы.
2. Постановку задачи в соответствии с темой лабораторной работы и согласование ее с руководителем.
3. Построение алгоритма решения задачи и его документирование в разделе «Краткие теоретические сведения» отчета.
4. Выполнение задания.
5. Подготовку отчета о выполненной работе и его защиту.

Структура отчета по проделанной работе:

1. Тема.
2. Цель занятия.
3. Постановка задачи.
4. Ход выполнения работы.
5. Блок-схема или псевдокод алгоритма решения задачи.
6. Текст программы.
7. Распечатка результатов.
8. Выводы.

Практическое занятие 1. Создание главного окна приложения в среде C#

Цель занятия: Изучить основные элементы среды разработки Visual Studio Integrated Development Environment (IDE - интегрированная среда разработки) C# при создании на языке C# приложений с графически интерфейсом.

Основные сведения

Среда разработки Visual Studio Integrated Development Environment (IDE) - интегрированная среда разработки) включает набор инструментов и не зависит от используемых языков программирования, представленных в Visual Studio. Visual Studio можно использовать для создания кода и на различных языках программирования: управляемый C++ - Manadged C++, Visual Basic.NET, Java.NET, C#.

В лабораторной работе проводится изучении среды разработки на языке программирования C# и следующих средств проектирования Windows - приложений:

основные окна среды разработки C#;

- построение базовой инфраструктуры с помощью Application Wizard (мастер создания приложений);
- использование дизайнера форм Dialog Painter (программа для рисования диалоговых окон) для оформления диалоговых окон;
- добавление новых функциональных возможностей в приложение с использованием вкладки Properties (свойства).

Обзор среды разработки C#

Для начала работы с Visual Studio.NET необходимо из главного меню выбрать пункт "Microsoft Visual Studio.NET" (VS). При этом на компьютере загрузится Developer Studio (визуальная среда разработки Microsoft Visual) на экране компьютера будет выведено окно, изображенное на рисунке 1.1.

Проектирование приложения

В качестве приложения разработаем простое приложение, пользовательский интерфейс которого будет содержать только главное окно. Для этого необходимо выполнить следующие шаги:

1. Создать рабочую область, называемую также рабочей средой (проектирования), рабочим пространством и рабочей обстановкой нового проекта.
2. Для создания каркаса приложения можно использовать мастер создания приложений - Application Wizard.
3. Изменить внешний вид автоматически создаваемых мастером окон до желаемого вида.

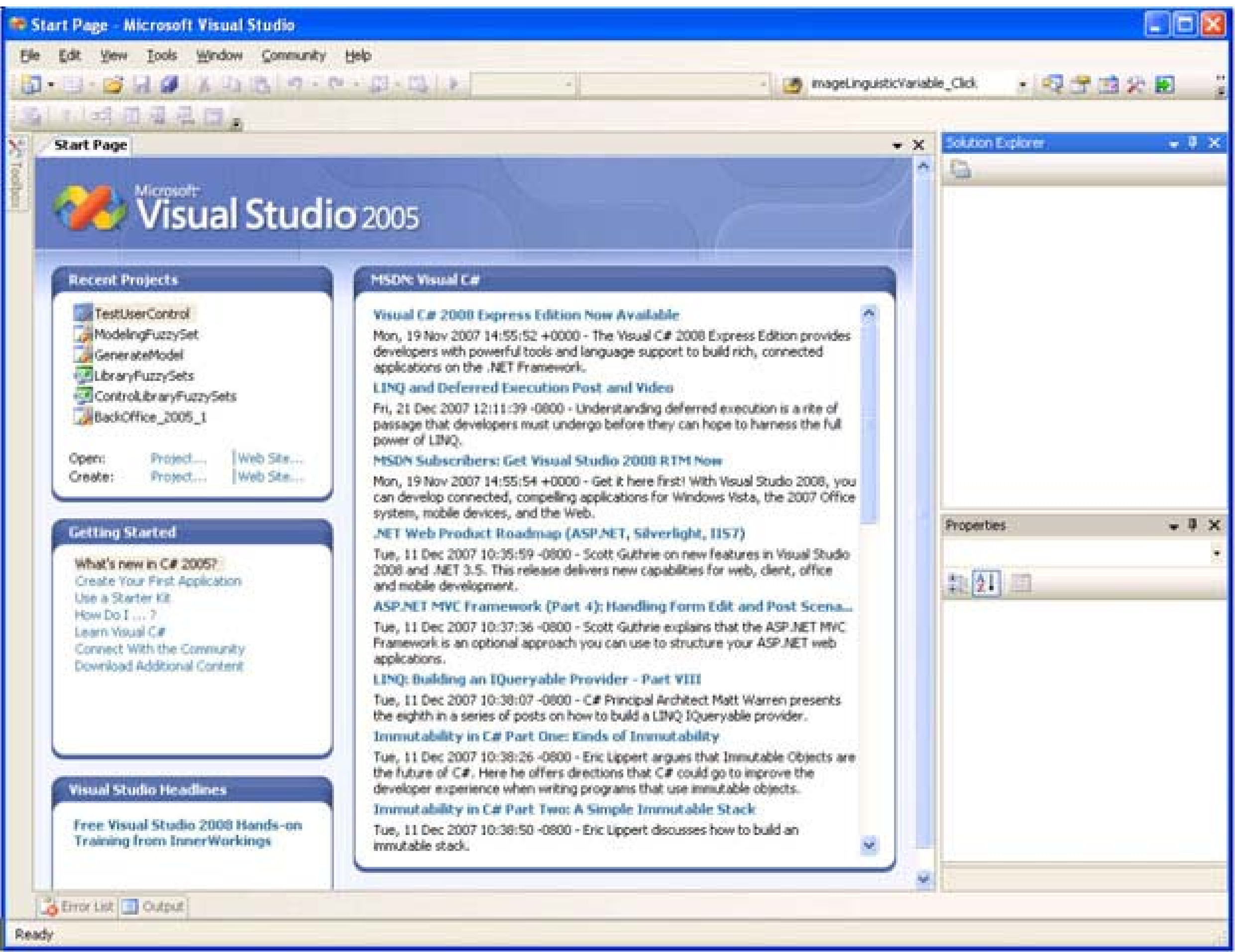


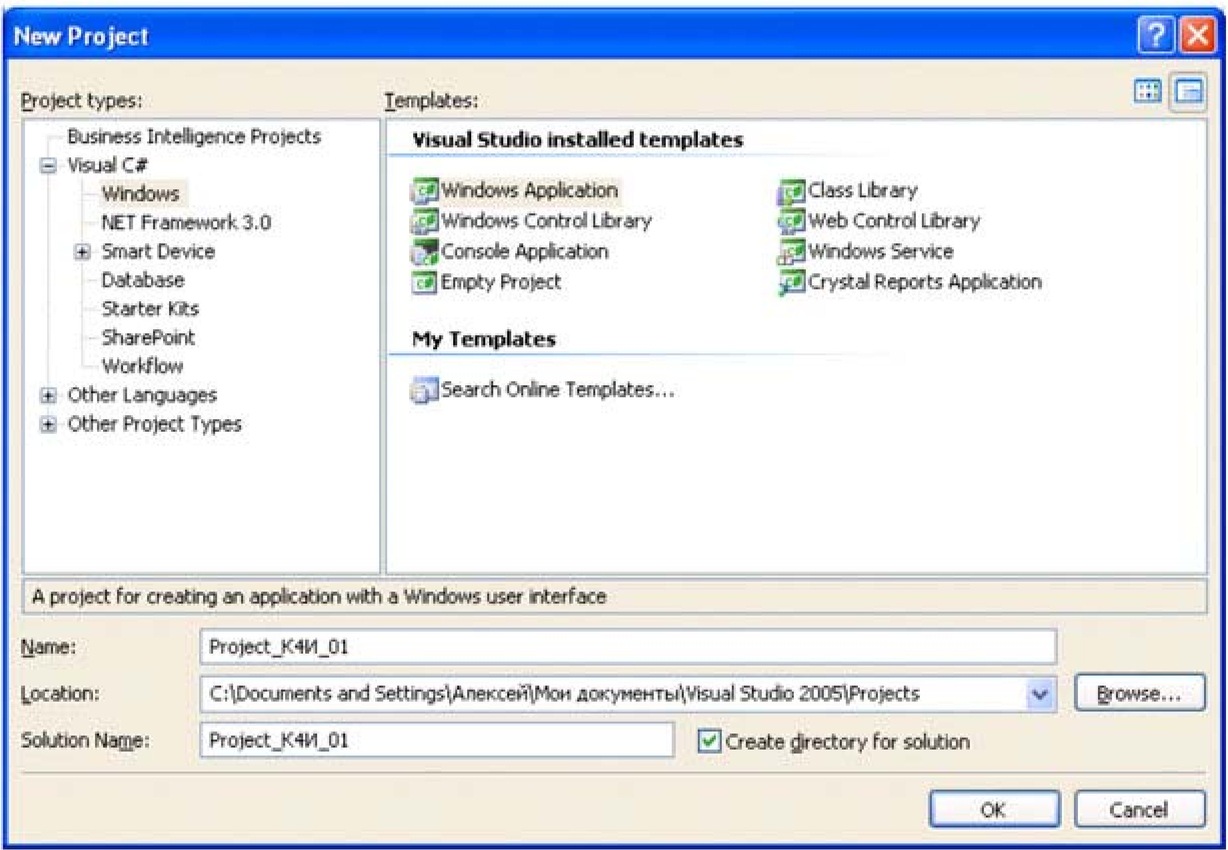
Рис. 1.1. Стартовое диалоговое окно IDE

4. Добавить код C#, который будет вызывать отображение приветствия.

Создание рабочей области проекта

В VS каждому разрабатываемому приложению нужна рабочая среда. Рабочая среда проекта состоит из папок, в которых хранятся файлы исходного кода, а также из папок, в которых хранятся различные конфигурационные файлы. Создание рабочей среды нового проекта производится следующим образом:

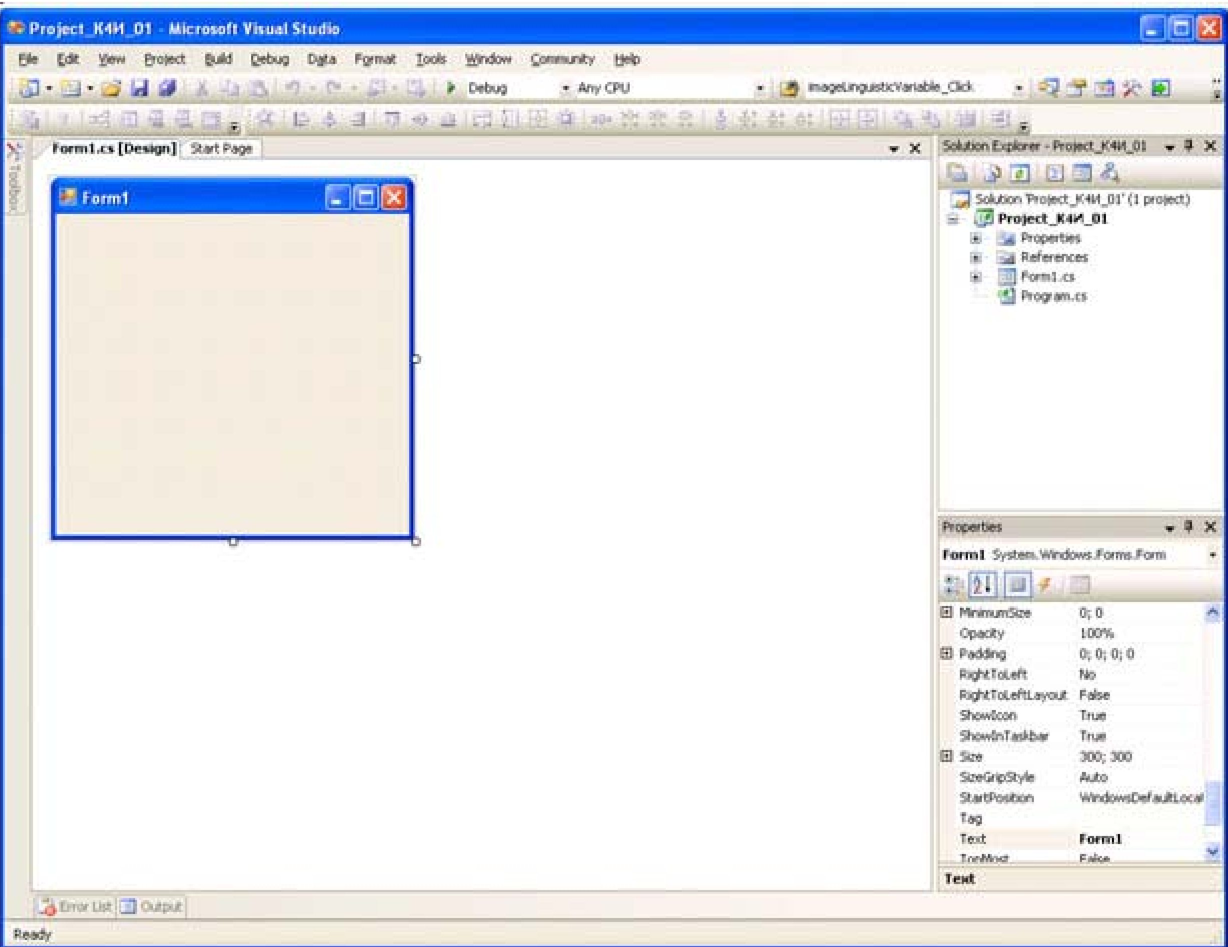
1. Щелкните на ссылке Project (Создать новый проект) метки Create на начальной странице (Start Page) VS.NET. При этом откроется окно создания нового проекта New Project (рисунок 1.2).



увеличить изображение

Рис. 1.2. Мастер создания нового проекта (New Project Wizard)

2. В дереве, отображаемом в подокне Project Type (Типы проектов) выберите "Visual C# /Windows". В подокне Templates (Шаблоны) выберите Windows Application (Приложение Windows).
3. В поле Name (Название проекта) наберите имя проекта - `Project_К4И_01` (имя проекта присваивается в соответствии со следующим синтаксисом: `Project_"номер группы"_"номер бригады в группе"`).
4. Щелкните на кнопке OK. (Да). Мастер создания нового проекта создаст новый класс `Form1`, производный от `System.Windows.Forms.Form` с правильно настроенным методом `Main( )`. В свойствах проекта автоматически будут созданы ссылки на необходимые сборки библиотеки базовых классов. На экране появится графический шаблон среды разработки (рисунок 1.3).

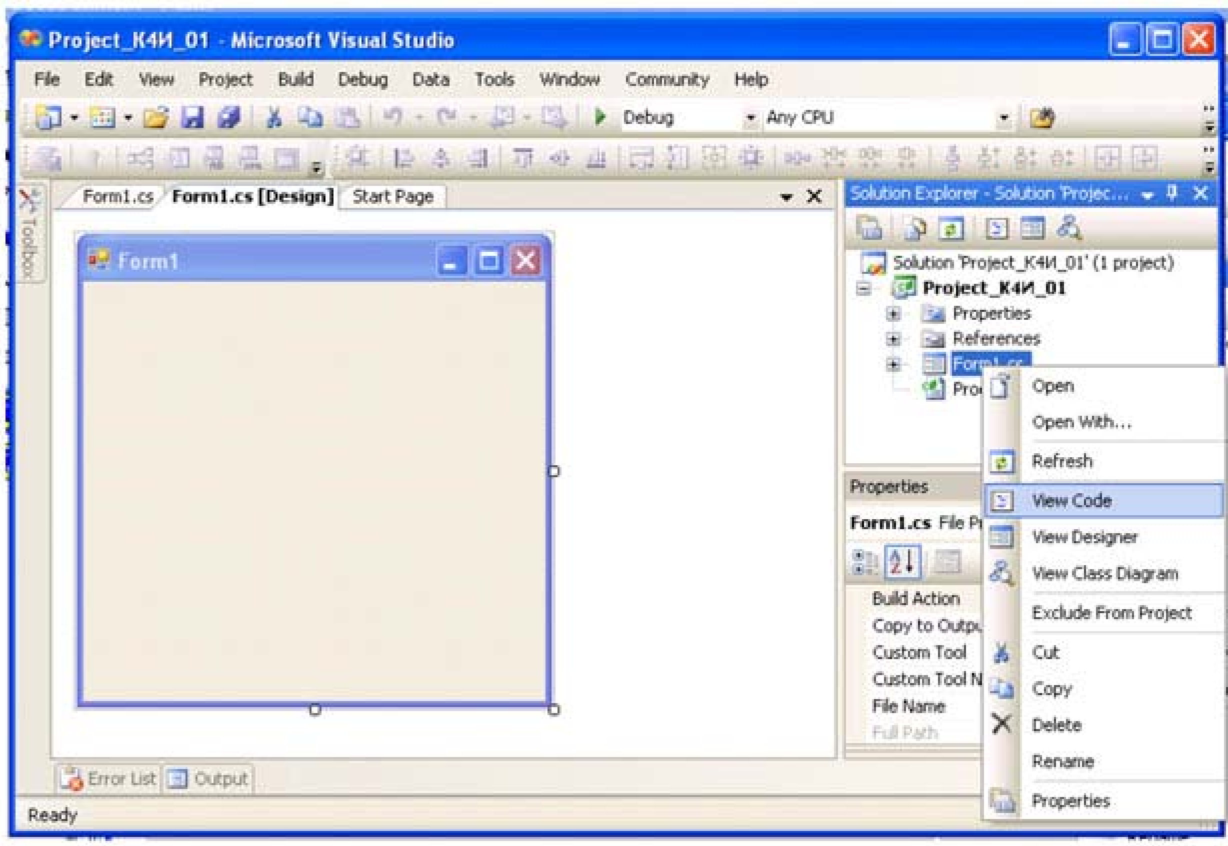


увеличить изображение

Рис. 1.3. Графический шаблон главного окна приложения

При помощи дизайнера графических форм можно добавлять в приложение любые элементы управления и он будет автоматически генерировать код для этих элементов (по умолчанию файл с главной формой приложения называется `Form1.cs`).

Для просмотра кода сгенерированного приложения можно в окне Solution Explorer щелкнуть правой кнопкой мыши на файле `Form1.cs` и в контекстном меню выбрать View Code (рисунок 1.4).



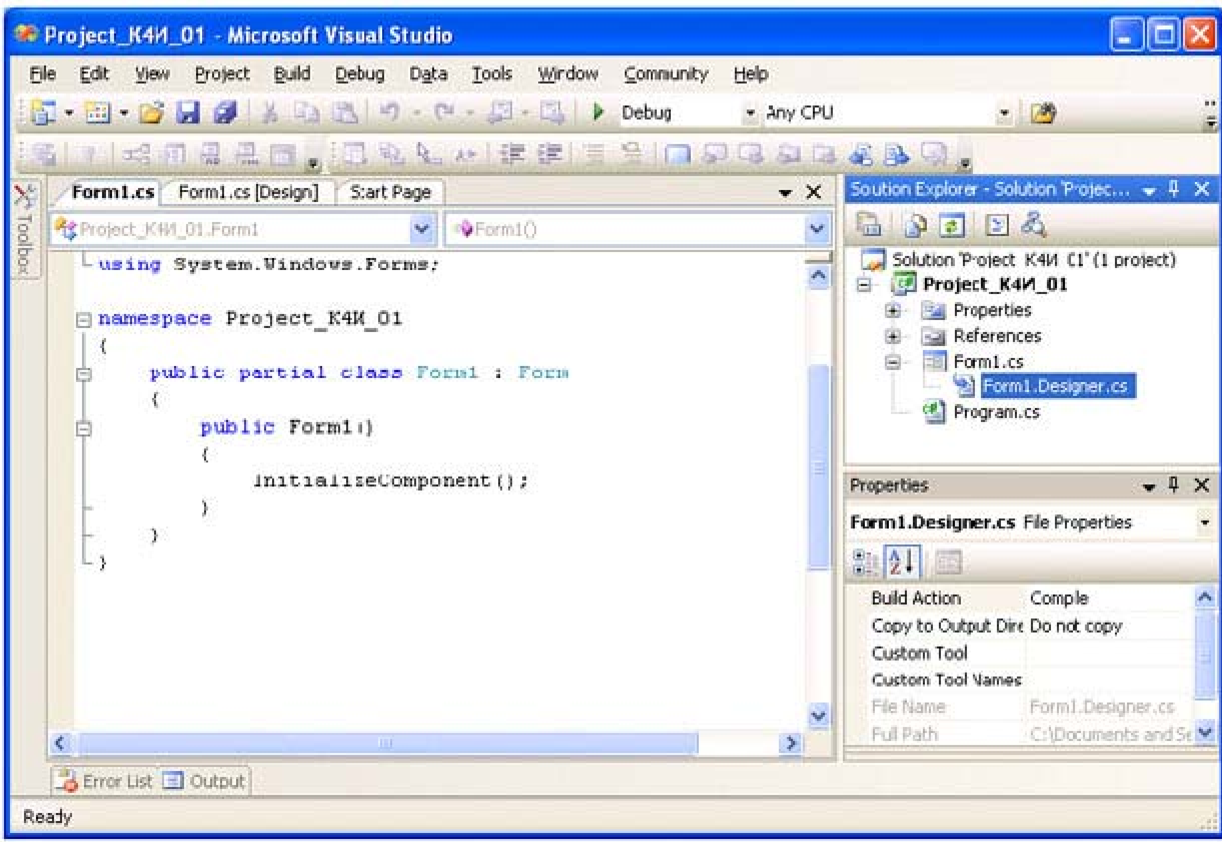
увеличить изображение

Рис. 1.4. Выбор режима View Code в контекстном меню

В результате будет выведен на экран следующий листинг кода приложения

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace Project_К4И_01
{
  public partial class Form1 : Form
  {
    public Form1()
   { InitializeComponent(); }
  }
}
```

Инициализация компонент реализуется кодом, который можно отобразить, в окне Solution Explorer щелкнуть на пункте Form.Designer.cs (рисунок 1.5).



увеличить изображение

Рис. 1.5. Выбор режима Form.Designer.cs

```
namespace Project_K4И_01
{
  partial class Form1
  {
    private System.ComponentModel.IContainer components = null;
    protected override void Dispose(bool disposing)
    {
       if (disposing && (components != null))
       {
          components.Dispose();
       }
       base.Dispose(disposing);
    }
    #region Windows Form Designer generated code
    private void InitializeComponent()
    {
       this.components = new System.ComponentModel.Container();
       this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
       this.Text = "Form1";
     }
     #endregion
  }
}
```

В определении класса Form1 используется ключевое слово partial, которое позволяет определять класс, структуру или интерфейс, распределенные по нескольким файлам. В Visual Studio 2005 классы Windows -форм формируются в двух файлах: Form1.cs и Form1.Designer.cs. В файле Form1.Designer.csприсутствует код, сгенерированный дизайнером Windows -формы, а файле Form1.cs - присутствует код инициализации класса и пользовательские члены класса (поля, свойства, методы, события, делегаты)

Код приложения (Program.cs) имеет следующий вид:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace Project_K4И_01
{
  static class Program
   {
     [STAThread]
     static void Main()
    {
     Application.EnableVisualStyles();
     Application.SetCompatibleTextRenderingDefault(false);
     Application.Run(new Form1());
    }
   }
}
```

Метод Main является точкой входа для приложения и вызывает Application.Run, который создает класс Form1.

Класс System.Windows.Forms.Application

Класс Application можно рассматривать как "класс низшего уровня", позволяющий нам управлять поведением приложения Windows Forms. Кроме того, этот класс определяет набор событий уровня всего приложения, например закрытие приложения или простой центрального процессора.

Наиболее важные методы этого класса (все они являются статическими) перечислены в таблице 1.1.

Таблица 1.1. Наиболее важные методы типа Application

Метод класса	**Назначение Application**
AddMessageFilter( )	Эти методы позволяют приложению перехватывать сообщения RemoveMessageFilter( ) и выполнять с этими сообщениями необходимые предварительные действия. Для того чтобы добавить фильтр сообщений, необходимо указать класс, реализующий интерфейс IMessageFilter
DoEvents( )	Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной

	операции. Можно сказать, что DoEvents() - это "быстрый и грязный" заменитель нормальной многопоточности
Exit( )	Завершает работу приложения
ExitThred( )	Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток
OLERequired( )	Инициализирует библиотеки OLE. Можете считать этот метод эквивалентом .NET для вызываемого вручную метода Olelnitialize( )
Run( )	Запускает стандартный цикл работы с сообщениями для текущего потока

Класс Application определяет множество статических свойств (таблице 1.2), большинство из которых доступны только для чтения.

Таблица 1.2. Наиболее важные свойства типа Application

Свойство	Назначение
CommonAppDataRegistry	Возвращает параметр системного реестра, который хранит общую для всех пользователей информацию о приложении
CompanyName	Возвращает имя компании
CurrentCulture	Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток
CurrentlnputLanguage	Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком
ProductName	Для получения имени программного продукта, которое ассоциировано с данным приложением
ProductVersion	Позволяет получить номер версии программного продукта

StartupPath	Позволяет определить имя выполняемого файла для работающего приложения и путь к нему в операционной системе

Многие из этих свойств предназначены для получения общей информации о приложении, такой как название компании, номер версии и т.п.

Таким образом, при помощи многих свойств (например, `CompanyName` или `ProductName`) можно очень просто получить метаданные уровня сборки. В сборке можно использовать любое количество встроенных и пользовательских атрибутов. В результате можно получить значение атрибута `[assembly:AssemblyCompany(" ")]` при помощи свойства `Application.CompanyName` без необходимости прибегать к использованию типов, определенных в пространстве имен `System.Reflection`.

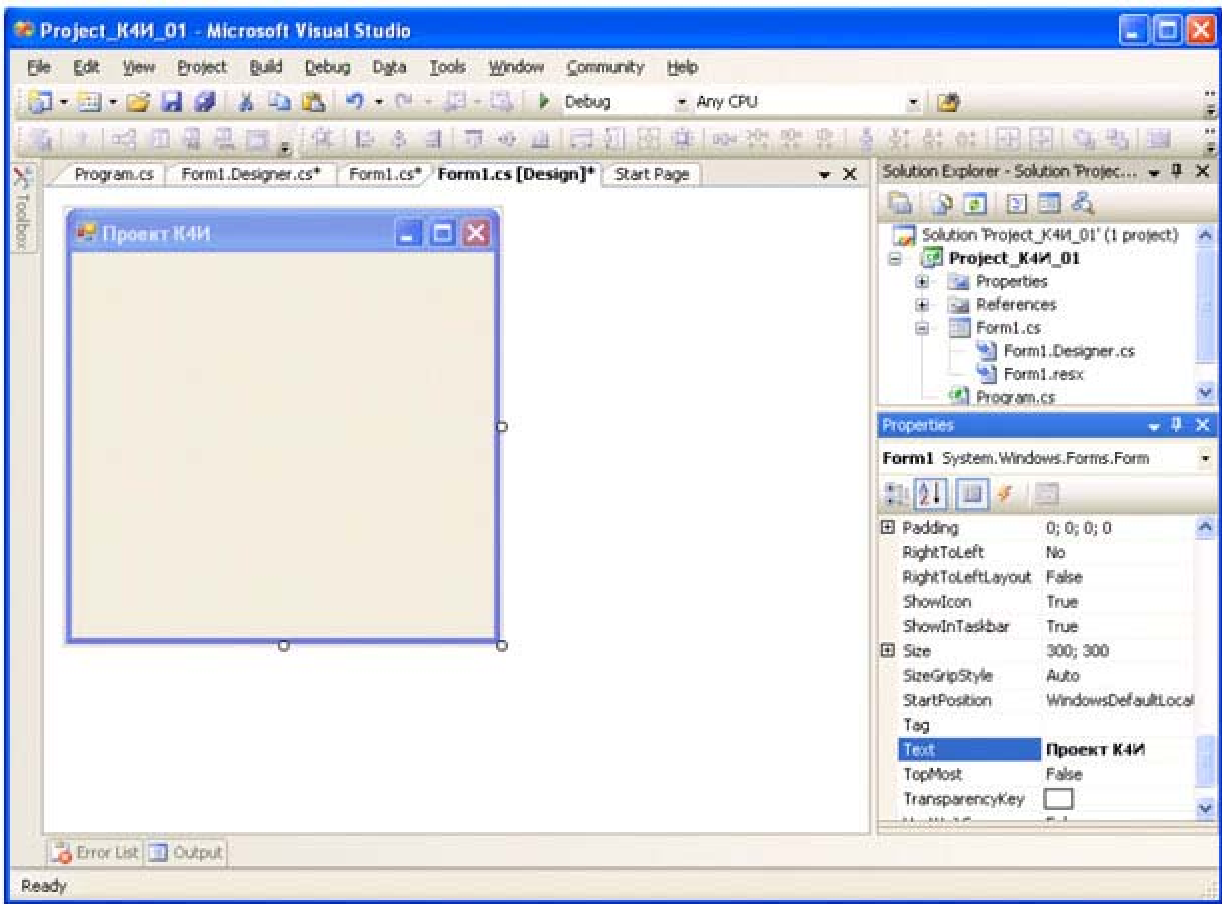
Проектирование окна приложения

Для разметки окон приложения в соответствии с требованиями пользователя необходимо изменить свойства класса `Forms1`. Это можно сделать с помощью дизайнера окон (Form Designer), путем изменения свойств в окне Свойства (Properties) или в коде программы.

Размеры окна можно изменить непосредственно в Form Designer с помощью мыши захватывая и, растягивая/сжимая границы окна.

Для изменения других свойств окна необходимо окно свойств Properties.

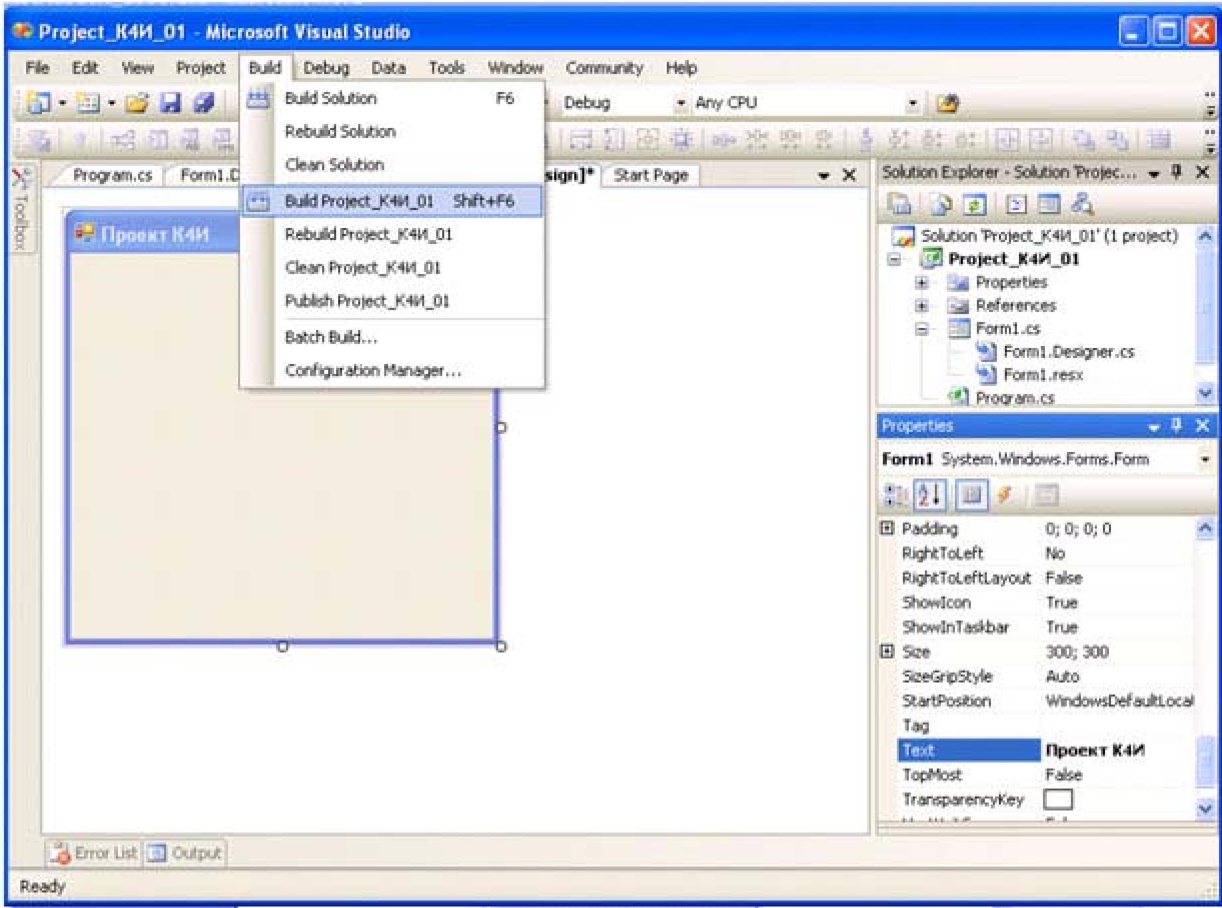
На вкладке Properties измените значение в поле Text (Заголовок) на Проект К4И. При этом на форме изменится заголовок окна (рисунок 1.6).



увеличить изображение

Рис. 1.6. Изменение значения в поле Text на вкладке Properties

Откомпилируйте приложение, выбрав из главного меню команду Build Project_К4И_01 (рисунок 1.7)



увеличить изображение

Рис. 1.7. Выбор из главного меню команды Build

В строке состояний должно появиться сообщение:

```
Build succeeded
```

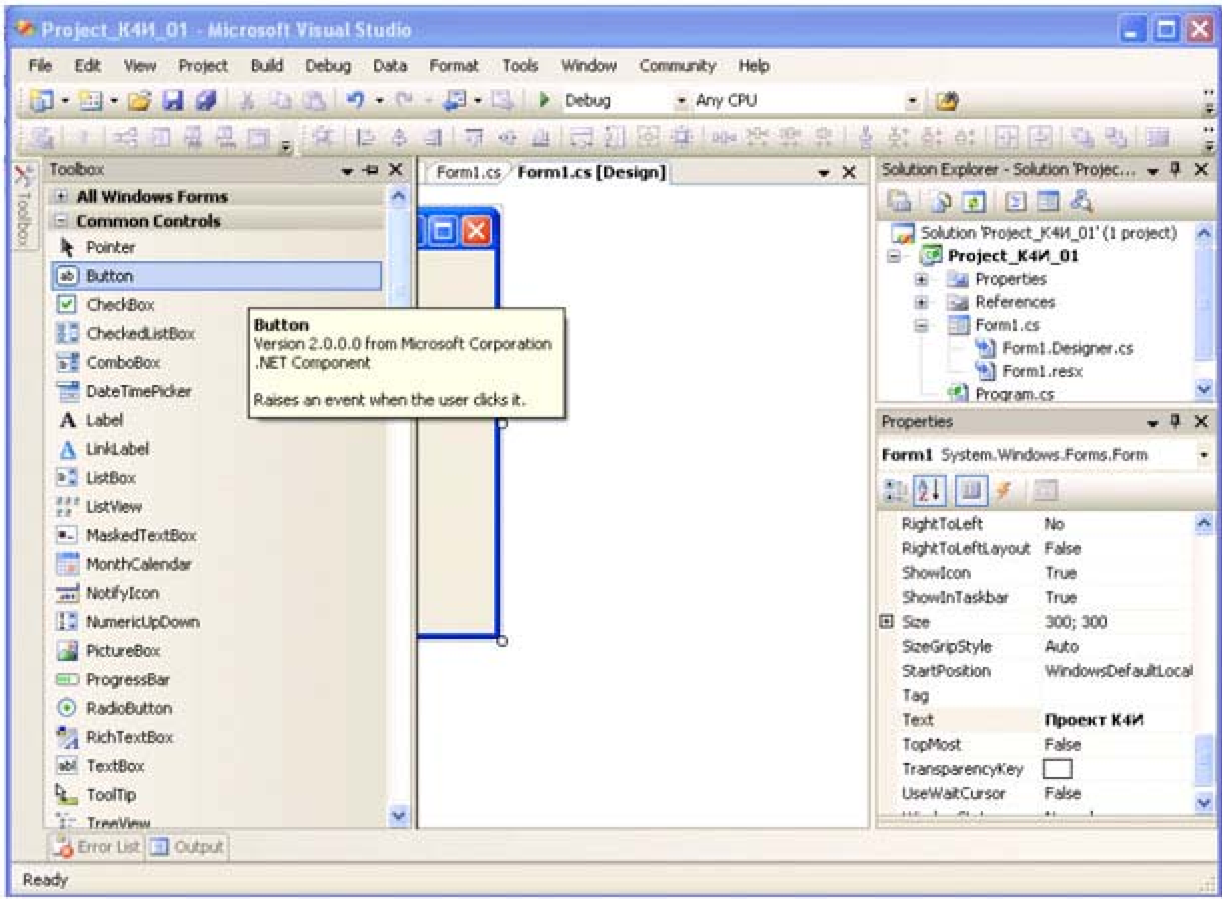
Для запуска приложения выберите из главного меню команду Debug/Start (F5). Приложение запустится в отладочном режиме и на экране появится разработанное окно (рисунок 1.8.).

Рис. 1.8. Окно приложения Project_К4И_01

Для закрытия окна щелкните мышью на кнопке

Добавление нового кода в приложение

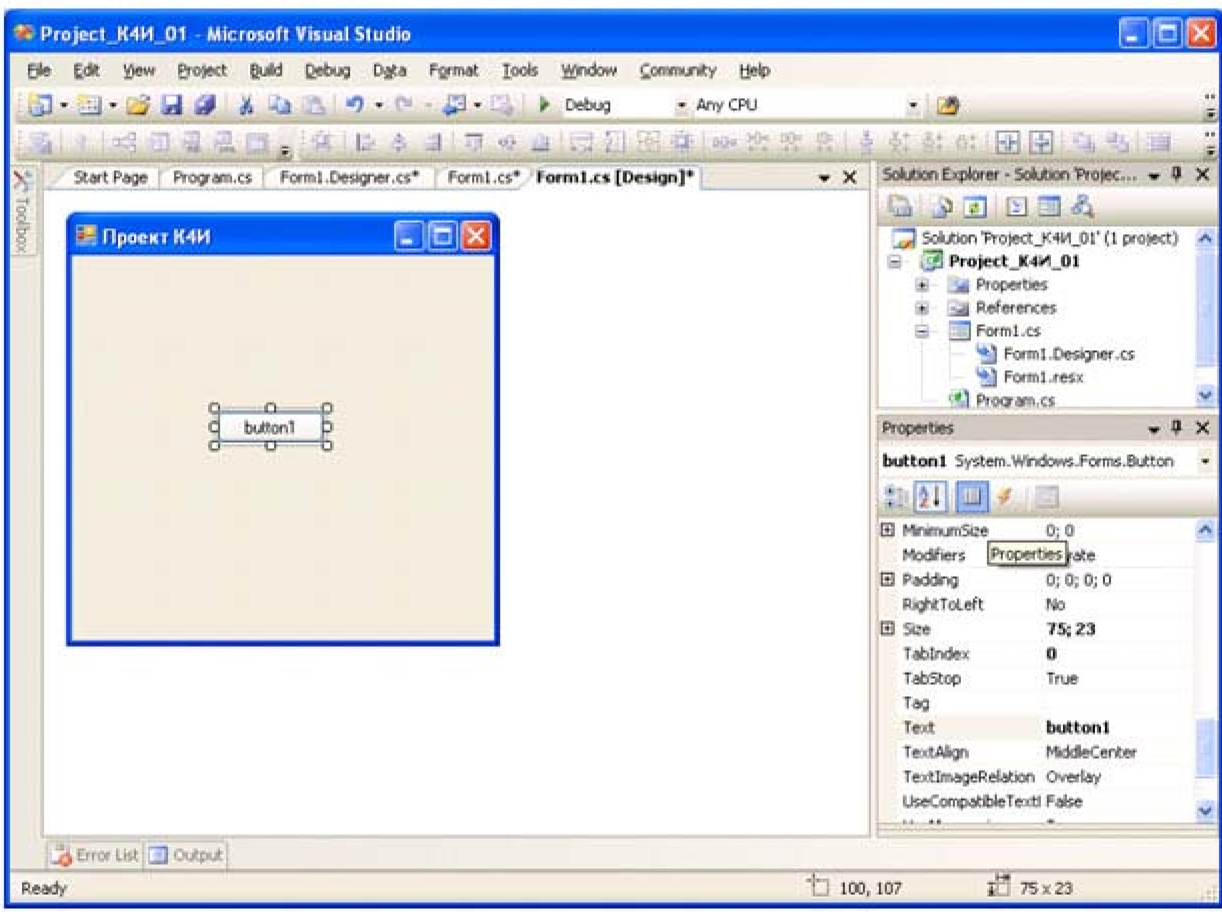
Добавим в главную форму элемент контроля - кнопку. Для этого откроем вкладку ToolBox (рисунок 1.9) и сначала щелкнем мышью на элементе Button вкладки, а затем щелкнем мышью на форме.



увеличить изображение

Рис. 1.9. Вкладка ToolBox

В результате получим форму с кнопкой (рисунок 1.10).

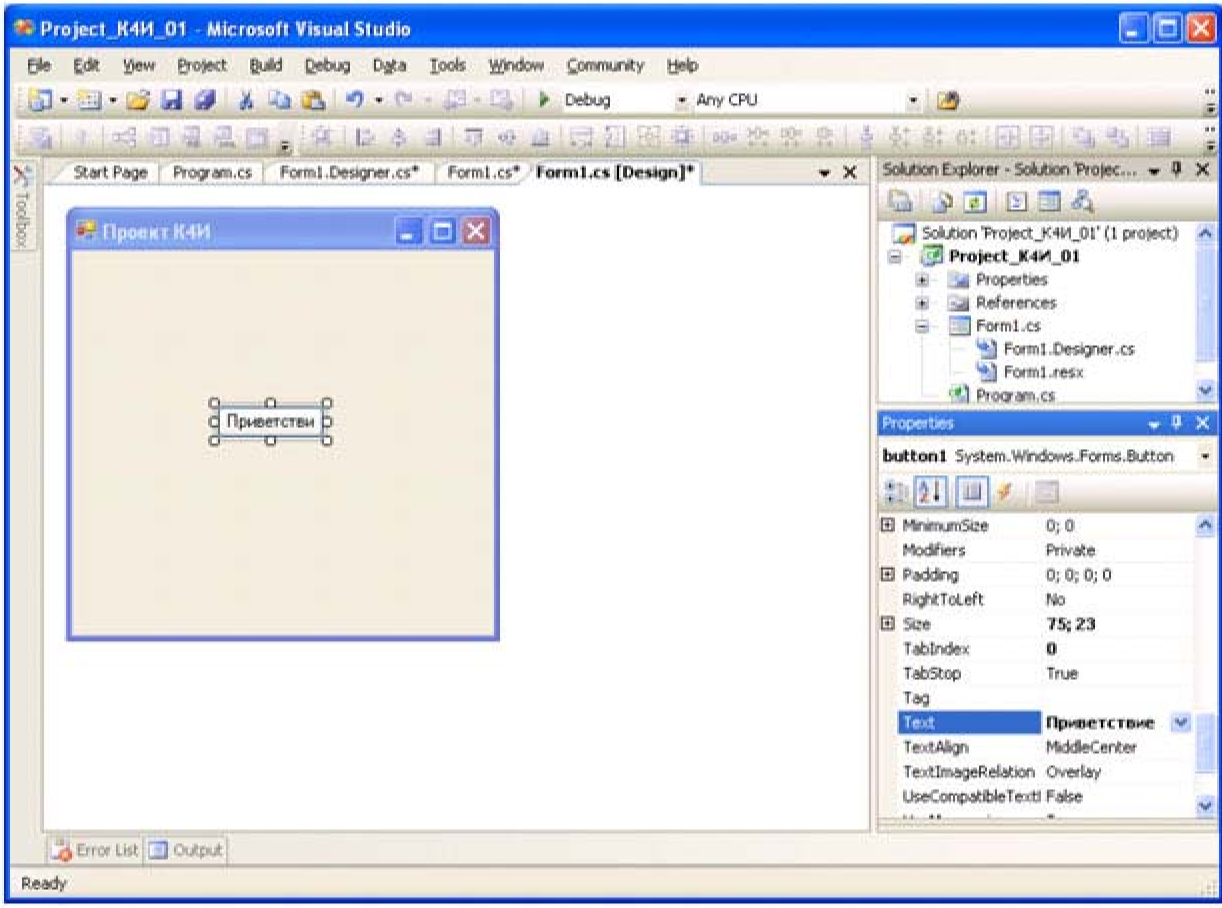


увеличить изображение

Рис. 1.10. Форма с установленной кнопкой

Установите кнопку в требуемое место на форме с помощью мыши.

Для задания текста на кнопке выделите ее на форме и откройте вкладку Свойства и измените свойство Text на "Приветствие". В результате название кнопки изменится (рисунок 1.11).



увеличить изображение

Рис. 1.11. Форма с измененным свойством Text кнопки

Для *связывания функций* кнопки с диалоговым окном необходимо создать обработчик события на нажатие кнопки. Для этого сделайте двойной щелчок на кнопке. В результате в коде приложения сформируется шаблон функции обработчика события `Click` для кнопки.

```
private void button1_Click(object sender, EventArgs e)
        {

        }
```

В полученный шаблон добавим функцию вывода диалогового окна с сообщением.

```
private void button1_Click(object sender, EventArgs e)
    {
     // Сообщение
     MessageBox.Show("Поздравляю с первым проектом на C#");
     }
```

После компиляции и запуска приложения получим следующее окно приложения (рисунок 1.12), а при нажатии кнопки будет выведено сообщение (рисунок 1.13).

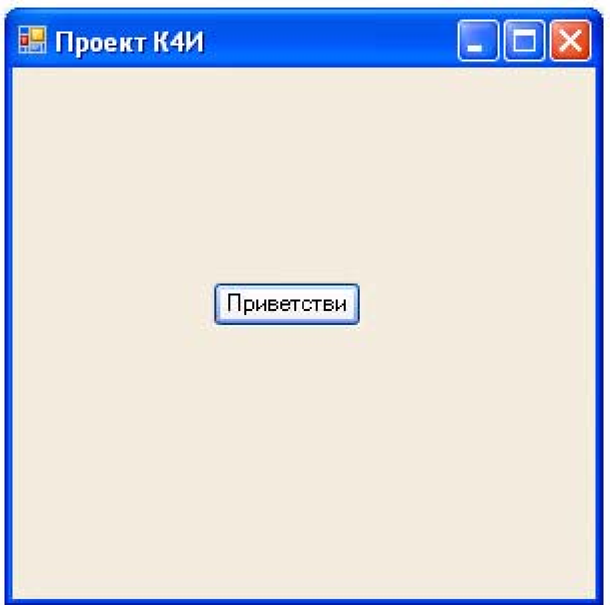


Рис. 1.12. Результат выполнения приложения

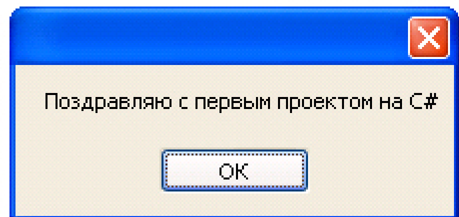


Рис. 1.13. Вывод сообщения

Структура и синтаксис функции

Рассмотрим код листинга функции:

```
private void button1_Click(object sender, EventArgs e)
    {
    // Сообщение
    MessageBox.Show("Поздравляю с первым проектом на C#");
    }
```

Первая строка является частью оболочки функции, сгенерированной Developer Studio на языке C#.

Первое слово в строке, `private` определяет видимость функции как внутреннюю, т.е. видимую только для членов класса `Form1`. Второе слово `void`, определяет тип данных возвращаемого значения (результата). Ключевое слово `void`, - перед именем функции, или в качестве аргумента функции (в скобках в конце строки), означает отсутствие соответствующего элемента. Третье слово в строке, `button1_Click`, обозначает имя функции. За именем функции следует список передаваемых ей аргументов, заключенный в круглые скобки. Круглые скобки нужно использовать всегда, даже когда у функции нет параметров.

Правило 1. В C# при вызове функции за ее именем должны стоять круглые скобки, даже если данной функции не передается ни один параметр.

В следующей строке листинга открывающая фигурная скобка (`{`) отмечает начало тела функции. В конце тела функции ставится закрывающая фигурная скобка (`}`).

Правило 2. Тело функции всегда заключается в фигурные скобки `{}`.

Следующая строка начинается с двух косых черт, или слешей (`//`). Все, что следует до конца строки после двух идущих подряд косых черт (без пробелов, табуляций и т.п. между ними) рассматривается компилятором как комментарий и игнорируется. Исключением являются строки, в которых косая черта является частью текстовой, или литерной строки (строки букв). Это один из способов комментирования кода. Второй способ чаще используется при добавлении в код нескольких строк комментариев. В этом случае начало комментария обозначается идущими подряд косой чертой и звездочкой (`/*`), а конец комментария завершается таким же набором символов, но переставленных в обратном порядке (`*/`).

Последняя строка в добавленном нами коде (в тексте это две строки):

```
MessageBox.Show("Поздравляю с первым проектом на C#");
```

Во-первых, C# чувствителен к регистру. В именах функций и переменных заглавные (прописные) буквы должны использоваться точно так же, как в их объявлениях. Это означает, что компилятор распознает следующие имена функций как имена трех различных функций:

```
MessageBox.Show   messageBox.Show    messagebox.Show
```

Правило 3. Язык C# чувствителен к регистру. При вводе программ, написанных на языке C#, учитывайте регистр. В частности, все идентификаторы вводите с учетом регистра.

За именем функции следуют аргументы функции, заключенные в круглые скобки, а после скобок стоит точка с запятой. Аргументы разделяются запятыми.

Задание на практическое занятие

1. Изучить теоретический материал.
2. Создать Windows форму.
3. На Windows форме создать кнопку "Приветствие".
4. Протестировать работу приложения
5. Добавить в форму две кнопки (1 и 2), для которых задать различные цвета (свойство `BackColor`).
6. Написать для кнопок 1 и 2 обработчики, которые изменяют цвета кнопок: при неоднократном нажатии любой кнопки цвета кнопок меняются (цвет кнопки 1 меняется на цвет кнопки 2 и наоборот).
7. Добавьте кнопку "Выход". Закрытие приложения обеспечивает метод `Exit( )` класса `Application`.
8. Протестировать работу приложения.

Практическое занятие 2. Создание главного меню приложения

Цель занятия: Изучить основные способы разработки главного *меню* приложении. Получить практические навыки в создании главного *меню* приложения.

Указания по использованию .NET

В любом языке программирования существуют традиционные стили программирования. Эти стили являются не частью самого языка, а соглашениями, скажем, по *именованию переменных* или использованию определенных классов, методов или функций. Если большинство разработчиков будут следовать одинаковым соглашениям, то им будет проще понять код друг друга, что, в свою *очередь*, облегчает поддержку программы. Так, общим соглашением в Visual Basic 6 было то, что строковые переменные должны иметь имена, начинающиеся с `s` или `str`, например `String sResult` или `String strMessage`. Однако соглашения зависят от языка и среды разработки. Программисты на C++ для платформы *Windows* традиционно используют *префикс* `psz` или `lpsz` для обозначения строк: `char *pszResult; char *lpszMessage;`. Но на Unix-машинах такие префиксы не применяются: `char *Result; char *Message;`.

В соответствии с соглашениями в C# имена переменных не должны иметь префиксов: `string Result; string Message;`.

Соглашение, согласно которому имена переменных содержат *префикс*, указывающий *тип данных*, известно как "венгерский" стиль именования объектов. При чтении такого кода разработчики могут сразу же сказать по имени переменной, какой *тип данных* она представляет.

В то время как для многих языков соглашения по именованию вырабатывались одновременно с развитием языка, для C# и платформы .NET Microsoft написала подробные рекомендации по использованию, которые приведены в документации MSDN для .NET/C#. Следовательно, с самого начала программы .NET будут иметь более высокий уровень совместимости по части понимания кода другими разработчиками. Эти рекомендации были разработаны с учетом опыта, полученного на протяжении более двадцати лет объектно-ориентированного программирования, и в результате являются тщательно продуманными и хорошо восприняты сообществом разработчиков.

Однако необходимо отметить, что рекомендации не то же самое, что спецификации языка. Рекомендаций следует придерживаться по мере возможности. Если имеется веская причина для их несоблюдения, это не будет проблемой. Отклонение от рекомендаций должно быть вызвано реальными причинами, а не простым нежеланием.

Одним из важных моментов является выбор имен для элементов программы: переменных, методов, классов, перечислений и пространств имен.

Очевидно, что названия обязаны отражать назначение элемента и не должны конфликтовать с другими именами.

Общая философия платформы .NET состоит в том, что *имя переменной* должно отражать назначение экземпляра переменной, а не *тип данных*.

Например, `Height` - хорошее название, а `IntegerValue` - нет. Однако этот принцип является труднодостижимым идеалом. В частности, при работе с элементами управления в большинстве случаев вам будет удобнее использовать имена переменных, подобные `ConfirmationDialog` и `ChooseEmployeeListBox`.

Конкретные рекомендации по именованию включают в себя следующие *разделы*.

Практически во всех случаях для имен следует использовать стиль Pascal, при котором первая буква каждого слова в названии является прописной

Например: `EmployeeSalary, ConfirmationDialog, PlainTextEncoding`.

Соединение слов с помощью знака подчеркивания не приветствуется, поэтому не придумывайте такие имена, как `employee_salary`. В других языках часто используют все прописные буквы в названиях констант. Это не рекомендуется в C#, поскольку такие имена трудно читать, лучше применять паскалевский стиль:

```
const int MaximumLength;
```

Еще одна рекомендуемая схема - именование в стиле camel. Именование camel аналогично паскалевскому стилю, за исключением того, что первая буква первого слова не является прописной: `employeeSalary, confirmationDialog, plainTextEncoding`.

Существуют две ситуации, в которых лучше применять такое именование. Имена всех параметров, передаваемых в методы, должны записываться в стиле camel:

```
public void RecordSale (string salesmanName,int guanuity);
```

Также можно использовать camel -соглашение для того, чтобы отличить два элемента, которые в противном случае имели бы одинаковые имена. Наиболее общий случай, когда свойство является оболочкой для поля.

```
private string employeeName;
public string EmployeeName
{ get
     { return employeeName; }
}
```

Приведенный код является совершенно корректным с точки зрения рекомендаций. Отметим, однако, что в этом случае следует применять соглашение camel для закрытых членов и соглашение Pascal для открытых или защищенных членов, чтобы другие классы, использующие ваш код, видели только имена в стиле Pascal(за исключением имен параметров).

В большинстве случаев следует применять соглашения Pascal. Тем не менее, соглашение camel рекомендуется для закрытых переменных, которые не видны вне класса, где две переменные имеют одинаковое назначение. Например, если есть `public` свойство, которое инкапсулирует `private` *поле* с тем же именем, то можно использовать соглашение camel для поля и соглашение Pascal для свойства, как в приведенном выше примере `EmployeeName`.

Также необходимо обращать внимание на чувствительность к регистру. C# чувствителен к регистру, поэтому синтаксически в C# допустимо, чтобы имена различались только регистром. Однако нужно помнить, что ваши сборки могут быть вызваны из приложений VB.NET, а VB.NET не является чувствительным к регистру. Поэтому использовать имена, отличающиеся только регистром, можно лишь в том случае, если они никогда не будут видны вне сборки. В противном случае код, написанный в VB.NET, не сможет корректно использовать вашу сборку.

Необходимо по возможности делать так, чтобы стиль всех имен совпадал. Например, если один из методов в классе называется `ShowConfirmationDialog`, то другому методу не следует давать имя `ShowDialogWarning` или `WarningDialogShow`. Он должен называться `ShowWarningDialog`.

Имена пространств имен следует выбирать особенно тщательно для того, чтобы избежать использования такого же имени, которое применяется где-то еще. Необходимо помнить, что .NET различает имена объектов в разделяемых сборках только по именам пространств имен. Если использовать для двух пакетов программного обеспечения одно и то же имя пространства имен и установить оба пакета на один *компьютер*, возникнут проблемы. Рекомендуется создавать *пространство имен* верхнего уровня с именем вашей компании, а затем вкладывать пространства имен, постепенно сужая их названия до технологии, группы или отдела, где вы работаете, или до названия пакета, для которого предназначены ваши классы. Microsoft рекомендует имена пространств имен, которые начинаются с `<НазваниеКомпании>.<НазваниеТехнологии>`, например,

WeaponsOfDestructionCorp.RayGunControllers

или

WeaponsOfDestructionCorp.Viruses.

Имена не должны конфликтовать с ключевыми словами. Если попытаться в программе назвать элемент по имени одного из ключевых слов C#, это практически всегда вызовет синтаксическую ошибку., так как *компилятор* предположит, что имя соответствует оператору.

Создание меню

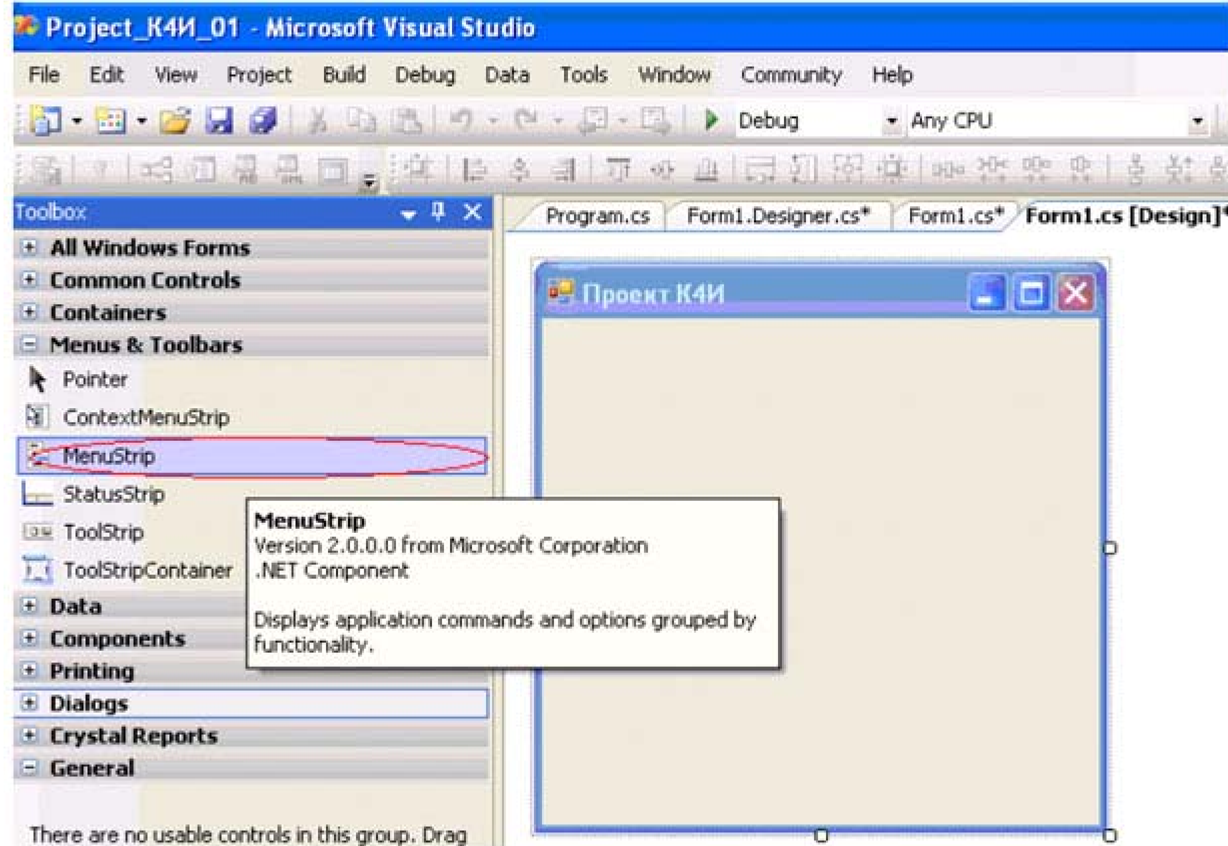
В пространстве имен System.Windows.Forms предусмотрено большое количество типов для организации ниспадающих главных *меню* (расположенных в верхней части формы) и контекстных *меню*, открывающихся по щелчку правой кнопки мыши.

Элемент управления `ToolStrip` представляет собой *контейнер*, используемый для создания структур *меню*, панелей инструментов и строк состояний.

Элемент управления `MenuStrip` - это *контейнер* для структур *меню* в приложении. Этот элемент управлении наследуется от `ToolStrip`. Система *меню* строится добавлением объектов `ToolStripMenu` к `menuStrip`.

Класс `ToolStripMenuItem` служит для построения структур *меню*.
Каждый *объект* `ToolStripMenuItem` представляет отдельный *пункт* в системе *меню*.

Начнем с создания стандартного ниспадающего *меню*, которое позволит пользователю выйти из приложения, выбрав *пункт* Объект > Выход. Для этого необходимо перетащить элемент управления `MenuStrip` (рисунок 2.1) на форму в конструкторе.



увеличить изображение

Рис. 2.1. Элемент управления MenuStrip

Элемент управления MenuStrip позволит вводить текст *меню* непосредственно в элементы *меню*. То, что должно получиться, представлено на рисунке 2.2.

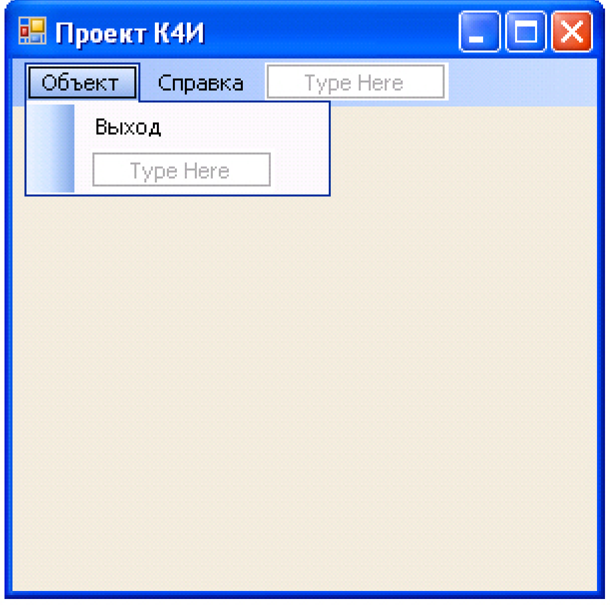


Рис. 2.2. Простое меню на форме

При помощи графических средств можно настроить свойства любого элемента *меню*. Для пункта *меню* "Объект" зададим свойство Name равным objektToolStripMenuItem, для пункта *меню* "Выход") - exitToolStripMenuItem, а для пункта *меню* "Справка" - HelpToolStripMenuItem.

При двойном щелчке на пункте *меню* "Выход" (*объект* exitToolStripMenuItem) Visual Studio автоматически сгенерирует оболочку для обработчика события Click и перейдет в окно кода, в котором нам будет предложено создать логику метода (в нашем случае exitToolStripMenuItem_Click):

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
    {
// Здесь мы определяем реакцию на выбор пользователем
// пункта меню
      }
```

На вкладке Свойства (Properties) при выводе окна событий, нажать кнопку событию Click будет соответствовать метод menuItemExit_Click (рисунок 2.3).

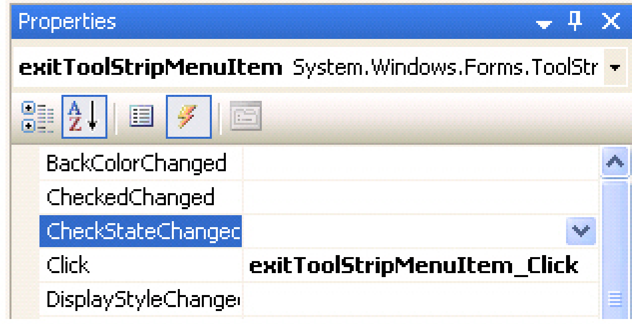


Рис. 2.3. Событие Click и обработчик события exitToolStripMenuItem_Click

Для корректного завершения приложения написать код для обработчика `exitToolStripMenuItem_Click`. Это можно сделать с помощью метода `Exit` класса `Application`:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
  {
    Application.Exit();
  }
```

Для тестирования созданного *меню* создадим обработчик для пункта *меню* "*Объект*", который будет сообщать, что выбран именно этот *пункт меню*.

```
private void objektToolStripMenuItem_Click(object sender, EventArgs e)
  {
    MessageBox.Show("Пункт меню Объект");
  }
```

При создании *меню* графическими средствами Visual Studio автоматически внесет необходимые изменения в служебный метод `InitializeComponent` и добавит переменные-члены, представляющие созданные элементы *меню*.

Задание на практическое занятие

1. Изучить теоретический материал.
2. Создать главное меню, включающее следующие пункты: "Объект", "Справочник", "Справка".
3. Для пункта "Объект" создать следующие подпункты: "Сотрудник", "Клиент", "Договор", "Поручение", "Сделка", "Выход".
4. Для пункта "Справочник" создать следующие подпункты: "Должность", "Страна", "Регион", "Город", "ИМНС".
5. Для пункта "Справка" создать подпункт - "О программе"
6. Протестировать работу приложения.

Практическое занятие 3. Создание многооконного приложения

Создание дочерней формы

Основа Интерфейса (MDI) приложения - MDI родительская форма. Это - форма, которая содержит MDI дочерние окна. Дочерние окна являются "подокнами", с которыми *пользователь* взаимодействует в MDI приложении. Создание MDI родительской формы описано в лабораторной работе 2.

Для определения главного окна (`Form1`), как родительской формы в окне Свойств, установите `IsMDIContainer` свойство - `true`. Это определяет форму как MDI *контейнер* для дочерних форм. Для того чтобы родительское окно занимало весь экран необходимо свойству `WindowsState` установить *значение*`Maximized`.

Создайте еще одно окно, которое будет дочерним (`FormEmployee`). Для этого выберите *пункт меню* Project/Add Windows Form.

Это окно должно вызываться из пункта главного *меню* "Сотрудник". Вставьте код, подобный следующему, чтобы создать новую MDI дочернюю форму, когда *пользователь* щелкает на пункте *меню*, например "Сотрудник" - *имя объекта* - `employeeToolStripMenuItem` (В примере ниже, *указатель* события обращается к событию `Click` для `employeeToolStripMenuItem_Click`).

```
private void menuItemEmploye_Click(object sender,
System.EventArgs e)
{ // Создать объект FEmployee класса FormEmployee
FormEmployee FEmployee = new FormEmployee();
        // Установить родительское окно для дочернего
        FEmployee.MdiParent = this;
        // Вывести на экран дочерее окно
        FEmployee.Show();
}
```

Данный обработчик приведет к выводу на экран дочернего окна.

Создание меню в дочерней форме

Добавьте в дочернее окно *пункт меню* "Действие" (`actionToolStripMenuItem`) с подпунктами "Отменить" (`undoToolStripMenuItem`), "Создать" (`createToolStripMenuItem`), "Редактировать" (`editToolStripMenuItem`), "Сохранить" (`saveToolStripMenuItem`) и "Удалить" (`removeToolStripMenuItem`). Перед пунктом удалить вставьте разделитель (`Separator - name = toolStripSeparator1`).

Добавьте в дочернее окно еще один *пункт меню* "Отчет" (`reportToolStripMenuItem`) *с* подпунктами "*По* сотруднику" (`reportToolStripMenuItem1`), "*По* всем сотрудникам" (`reportToolStripMenuItem2`). Дочернее окно будет иметь вид, представленный на рисунке 3.1

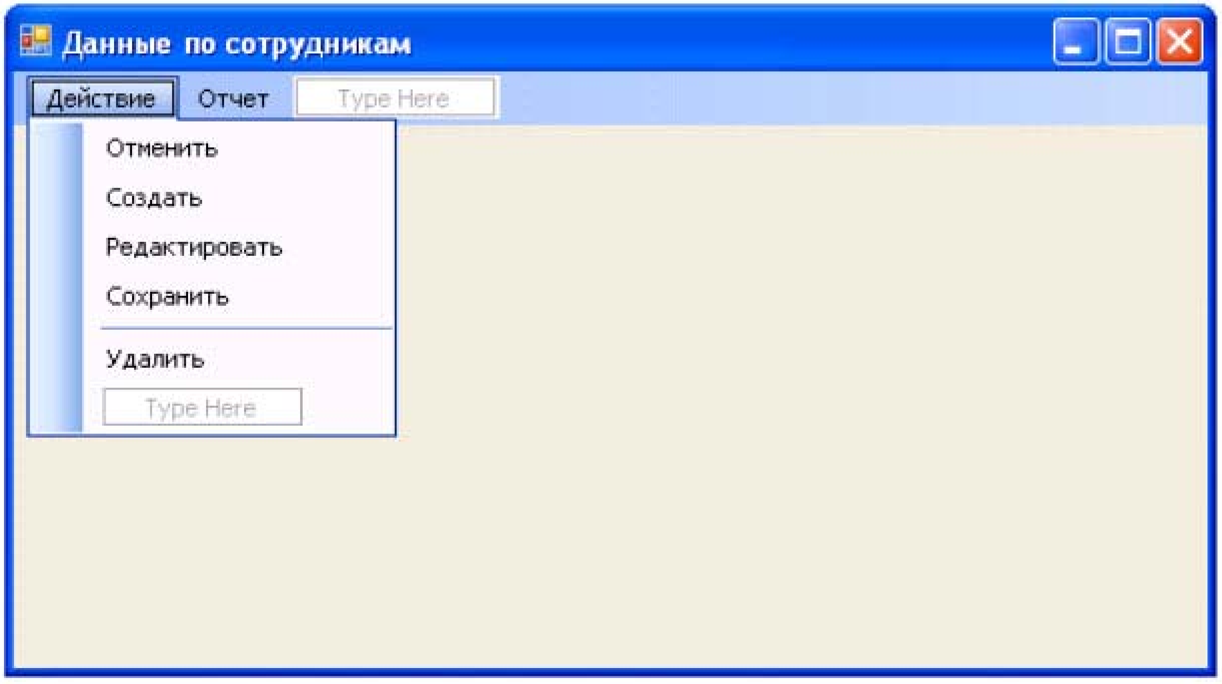


Рис. 3.1. Дочернее окно с меню

В главном *меню* родительской формы (Form1) имеются пункты "*Объект*", "Справочник" и "Справка". В дочерней форме (FormEmployee) сформированы пункты *меню* "Действие" и "Отчет". При загрузке дочерней формы *меню* родительской и дочерних форм должны были объединены и составлять следующую последовательность: "*Объект*", "Действие", "Отчет", "Справочник" и "Справка". *Объединение* пунктов *меню* производится с помощью задания значений свойств MergeAction и MergeIndex для объектов ToolStripMenuItem.

Проверьте, чтобы в *меню* главного окна для объекта objectToolStripMenuItem свойство MergeAction было установлено Append, а MergeIndex было равно 0, а для объектов dictionaryToolStripMenuItem и helpToolStripMenuItem - соответственно 1 и 2. С учетом этого, в окне "Сотрудник" для объектовactionToolStripMenuItem (Действие) и "Отчет" (reportToolStripMenuItem) свойству MergeAction необходимо задать *значение* Insert, а свойству MergeIndex задаем порядковый номер который определяет позицию данного пункта *меню* обновленном главном *меню*, т.е. 1 (после объектаobjectToolStripMenuItem).

После компиляции программы, запуска ее на выполнение и вызова пункта *меню* "Сотрудник" экран должен иметь вид, представленный на рисунке 3.2.

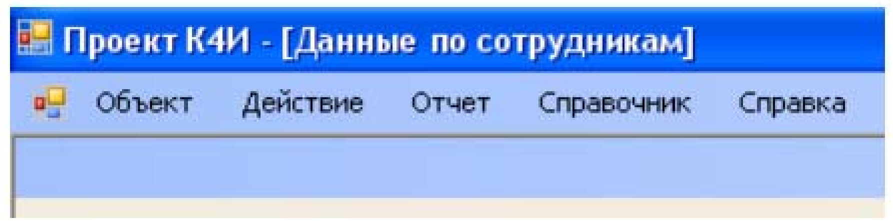


Рис. 3.2. Дочернее окно с подключенным меню

Создание обработчиков для меню дочерней формы

Созданные пункты *меню* для дочернего окна должны инициировать выполнение соответствующих функций (Отменить, Создать, Редактировать, Сохранить и Удалить) приложения в отношении объектов конкретного дочернего окна. Для дочернего окна "Данные *по* сотруднику" эти функции должны выполнять соответственно

отмену редактирования данных *по* сотруднику (*функция* "Отменить"), создавать новые данные *по* сотруднику (*функция* "Создать"), редактировать данные *по* сотруднику (*функция* "Редактировать"), сохранять созданные вновь или отредактированные *функция по* сотруднику (*функция* "Сохранить") и удалять данные *по* сотруднику (*функция* "Удалить").

Описанную функциональность целесообразно реализовать в программе в виде методов класса созданного `FormEmployee`. В приложении необходимо создать следующие методы:

- `Undo` - отменить;
- `New` - создать;
- `Edit` - редактировать;
- `Save` - сохранить;
- `Remove` - удалить.

На начальных этапах проектирования, как правило, неясна реализация каждого метода, поэтому целесообразно их выполнять в виде методов-заглушек, которые только сообщают пользователю о своем вызове, а в дальнейшем необходимо написать реальный код.

Для создания метода `Undo` в коде файла `FormEmployee.cs` добавьте следующий метод:

```
private void Undo( )
{  MessageBox.Show("метод Undo"); }
```

Далее создаем обработчик события вызова пункта *меню* "Отменить". Для этого в дизайнере формы класса `FormEmployee` делаем *двойной щелчок* на пункте *меню* "Отменить". Инструментальная среда VS сгенерирует следующий код:

```
private void undoToolStripMenuItem_Click(object sender, EventArgs e)
   {
   }
```

В код обработчика `undoToolStripMenuItem_Click` добавим *вызов метода* `Undo`:

```
private void undoToolStripMenuItem_Click(object sender, EventArgs e)
  {
     Undo();
  }
```

Откомпилируем *приложение* и протестируем *вызов метода* `Undo`. В результате выбора пункта *меню* "Отменить" должно быть выведено *диалоговое окно* с сообщением, приведенным на рисунке 3.3.

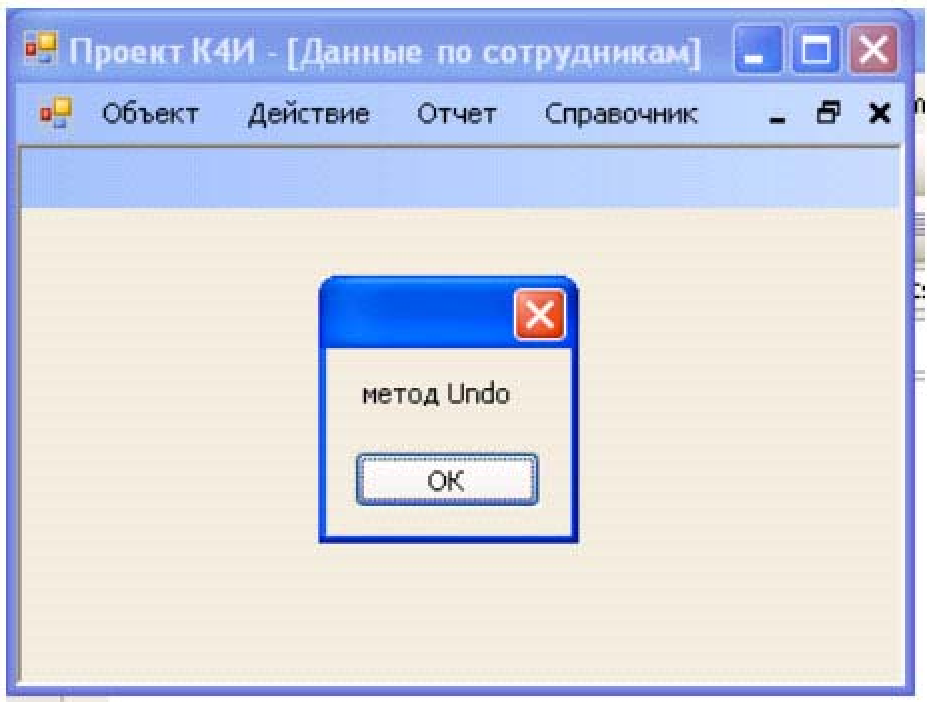


Рис. 3.3. Дочернее окно с подключенным меню

Аналогичным образом создайте методы-заглушки для функций "Создать", "Редактировать", "Сохранить" и "Удалить".

Задание на практическое занятие

1. Изучить теоретический материал.
2. Создать дочернее окно.
3. В дочернее окно добавить пункты меню.
4. Написать обработчик для вызова из главного меню дочернего окна.
5. Создать коды методов-заглушек для функций приложения.
6. Создать обработчики для вызова пунктов меню.
7. Протестировать работу приложения.

Практическое занятие 4. Создание пользовательских диалоговых окон

Цель занятия: Изучить основные способы построения диалоговых окон, их параметры и получить практические навыки в разработке

Основные сведения

Диалоговое окно - это форма, обладающая некоторыми специальными характеристиками. Первая отличительная черта большинства диалоговых окон - то, что их размер изменять нельзя. Кроме того, в диалоговых окнах обычно не используются *элементы управления*, помещаемые в верхнюю часть обычных форм:`ControlBox`, `MinimizeBox` и `MaximizeBox`. Для пользователя *диалоговое окно* в противоположность обычному является практически неизменяемым.

Диалоговые окна бывают модальные и немодальные. Если *приложение* открывает модальное окно, то работа приложения блокируется до тех пор, пока не будет закрыто модальное окно. Немодальные окна могут работать одновременно с породившим их главным окном приложения. Такие окна часто используются для "плавающих" инструментальных панелей, настройки различных параметров приложения, причем отсутствие модальности позволяет использовать в приложении измененные параметры, не закрывая окна настройки этих параметров.

Простейшее модальное *диалоговое окно* можно создать на базе класса `MessageBox`, входящего в библиотеку Microsoft .NET Framework. В лабораторной работе 3 иллюстрировалось применение простейшего модального диалогового окна для вывода сообщения об активизации метода `Undo`. Для отображения диалогового окна использовался метод `Show`, передав ему через *параметр* текст сообщения `"метод Undo"`. Прототип использованного метода `Show` следующий:

```
public static DialogResult Show(string message);
```

Когда *пользователь* щелкает кнопку OK, метод `Show` возвращает *значение*, равное `DialogResult.OK`

Существует множество перегруженных вариантов метода `MessageBox.Show`, позволяющих задать необходимый внешний вид диалоговой панели, а также количество и тип расположенных на ней кнопок.

Прототип наиболее общего варианта метода `MessageBox.Show`, позволяющий реализовать практически все возможности диалогового окна MessageBox, приведен ниже

```
public static DialogResult Show
{
string message, //  текст сообщения
string caption, //  заголовок окна
MessageBoxButtons btns, // кнопки,  расположенные в окне
MessageBoxIcon icon, //  значок, расположенный в окне
MessageBoxDefaultButton  defButton, // кнопка по умолчанию
MessageBoxOptions  opt // дополнительные параметры
};
```

Параметр `caption` позволяет задать текст заголовка диалогового окна MessageBox. С помощью параметра `btns` можно указать, какие кнопки необходимо расположить в окне диалогового окна. Этот *параметр* задается константами из перечисления `MessageBoxButtons` (таблица 4.1)

Таблица 4.1. Перечисление MessageBoxButtons	
Константа	**Кнопки, отображаемые в окне MessageBox**
OK	OK
OKCancel	OK, Cancel
YesNo	Yes, No
YesNoCancel	Yes, No, Cancel
RetryCancel	Retry, Cancel
AbortRetryIgnore	Abort, Retry, Ignore

Параметр icon метода MessageBox.Show позволяет выбрать один из нескольких значков для расположения в левой части диалогового окна. Он задается в виде *константы* перечисления MessageBoxIcon (таблица 4.2).

Таблица 4.2. Перечисление MessageBoxIcon	
Константа	**Значок**
Asterisk, Information	
Error, Stop	
Exclamation, Warning	
Question	

None	Значок не отображается

Параметр defButton метода MessageBox.Show предназначен для выбора кнопки, которая получит фокус сразу после отображения диалогового окна. Этот *параметр* должен иметь одно из значений перечисления MessageBoxDefaultButton (таблица 4.3).

Таблица 4.3. Перечисление MessageBoxDefaultButton

Константа	Номер кнопки, получающей фокус ввода по умолчанию
Button 1	1
Button 2	2
Button 3	3

Если в диалоговом окне отображаются кнопки Yes, No и Cancel, то они будут пронумерованы последовательно: кнопка Yes получит номер 1 (константа Button1), кнопка No - номер 2 (константа Button2), а кнопка Cancel - номер 3 (константа Button3).

Параметр opt метода MessageBox.Show позволяет задать дополнительные параметры. Эти параметры должны иметь значения из перечисления MessageBoxOptions (таблица 4.4).

Таблица 4.4. Перечисление MessageBoxOptions

Константа	Описание
DefaultDesktopOnly	Окно с сообщением отображается только на рабочем столе, выбранном по умолчанию
RightAlign	Текст сообщения выравнивается по правому краю диалогового окна
RtlReading	Текст отображается справа налево
ServiceNotification	Окно отображается на активном рабочем столе, даже если к системе не подключился ни один пользователь. Данный параметр предназначен для приложений, работающих как сервисы ОС Microsoft

	Windows

Если в окне диалогового окна MessageBox имеется несколько кнопок, то для того, что бы определить, какую кнопку щелкнул *пользователь*, *программа* должна проанализировать *значение*, возвращенное методом `MessageBox.Show`.

Метод `MessageBox.Show` может вернуть одно из нескольких значений перечисления `DialogResult` (таблица 4.5).

Таблица 4.5. Перечисление DialoqResult

Константа	Кнопка, при щелчке которой возвращается эта константа
`Abort`	Abort
`Cancel`	Cancel
`Ignore`	Ignore
`No`	No
`None`	Модальное диалоговое окно продолжает работать
`OK`	OK
`Retry`	Retry
`Yes`	Yes

Изменим метод `Remove`, добавив в него предупреждение перед удалением данных *по* сотруднику. Текст кода метода `Remove` должен иметь следующий вид:

```
private void Remove()
{
    DialogResult result = MessageBox.Show(" Удалить данные  \n по сотруднику?
",
      "Предупреждение", MessageBoxButtons.YesNo, MessageBoxIcon.Warning,
        MessageBoxDefaultButton.Button2);
    switch (result)
```

```
    {
        case DialogResult.Yes:
            {
//выполнить действия по удалению данных по сотруднику
                MessageBox.Show("Удаление данных");
                break;
            }
        case DialogResult.No:
            {
//отмена удаления данных по сотруднику
                MessageBox.Show("Отмена удаления данных");
                break;
            }
    }
```

В результате исполнения кода приложения и выбора пункта *меню* "Удалить" будет выводиться предупреждение, приведенное на рисунке 4.1 .

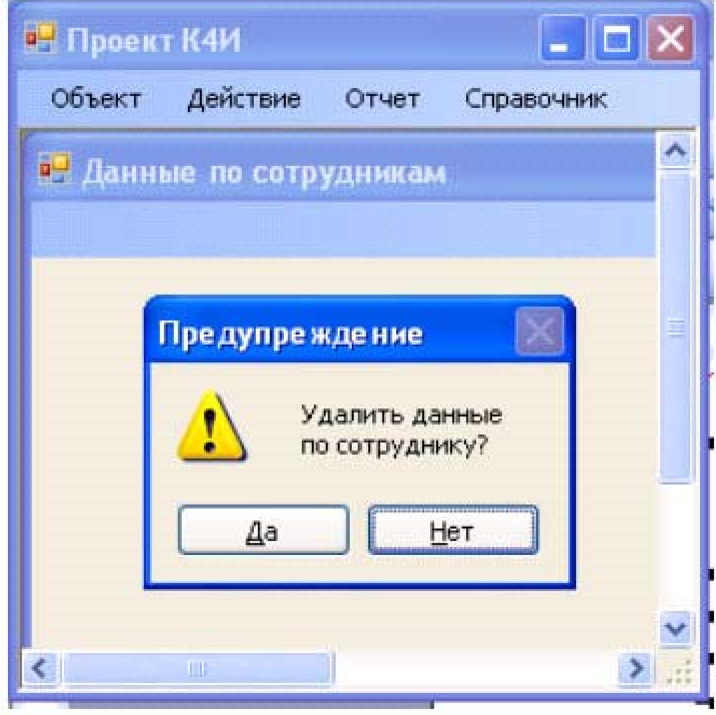


Рис. 4.1. Модальное диалоговое окно предупреждения

Диалоговое окно можно создать не только на основе класса `MessageBox`, но и с использованием Windows - формы.

Создадим новую форму FormAbout для вывода справочной информации о разрабатываемом приложении, которое должно иметь вид представленный на рисунке 4.2 .

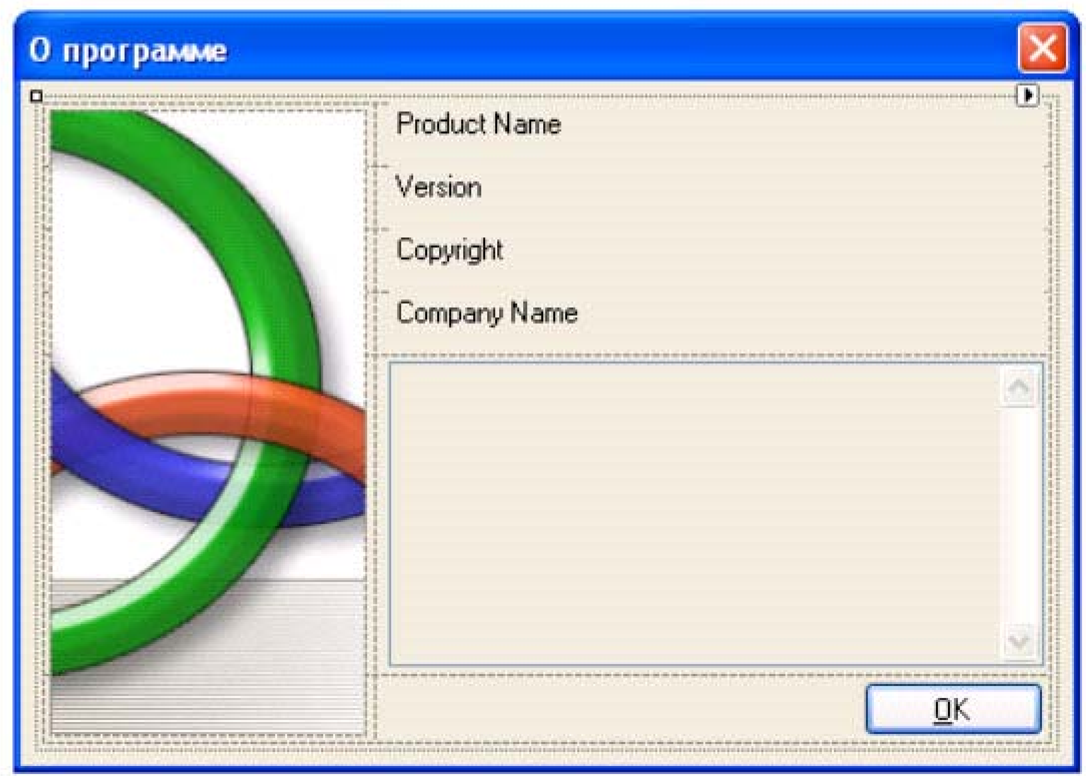


Рис. 4.2. Общий вид Windows - формы FormAbout

Для этого добавим в проект новый *компонент* (рисунок 4.3), выбрав из списка `AboutBox` (рис. 4.4).

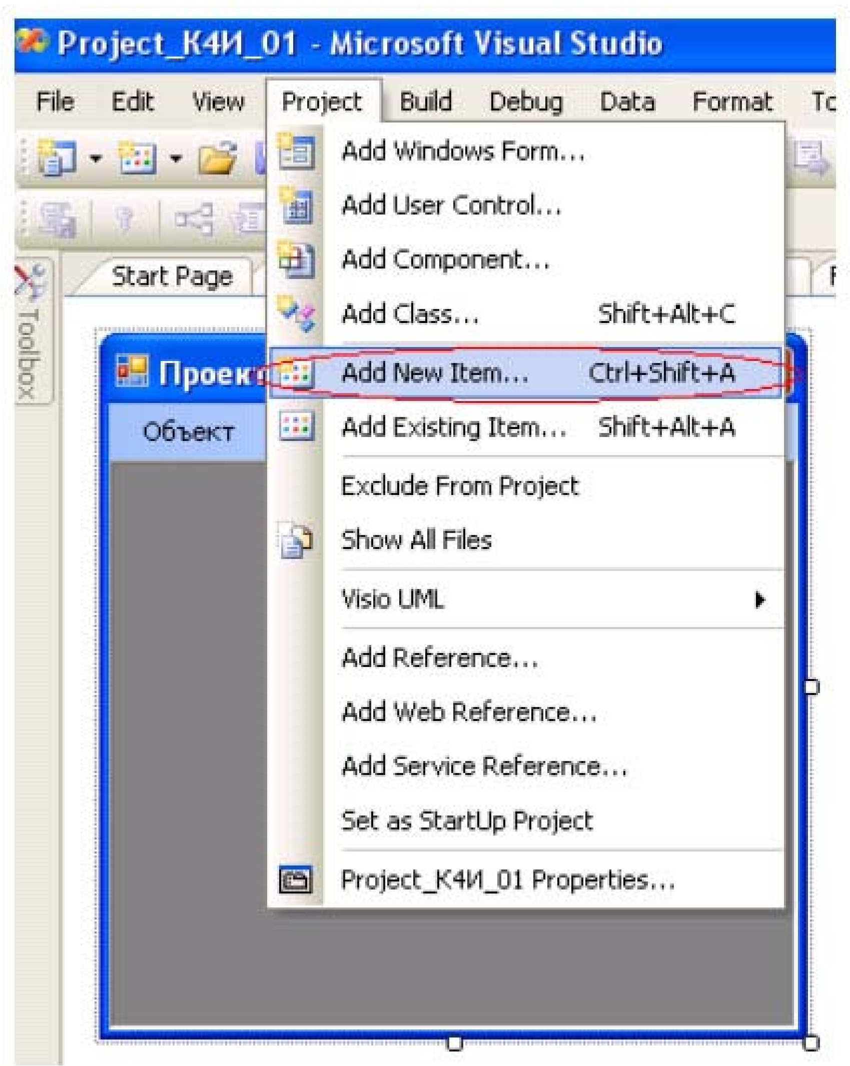
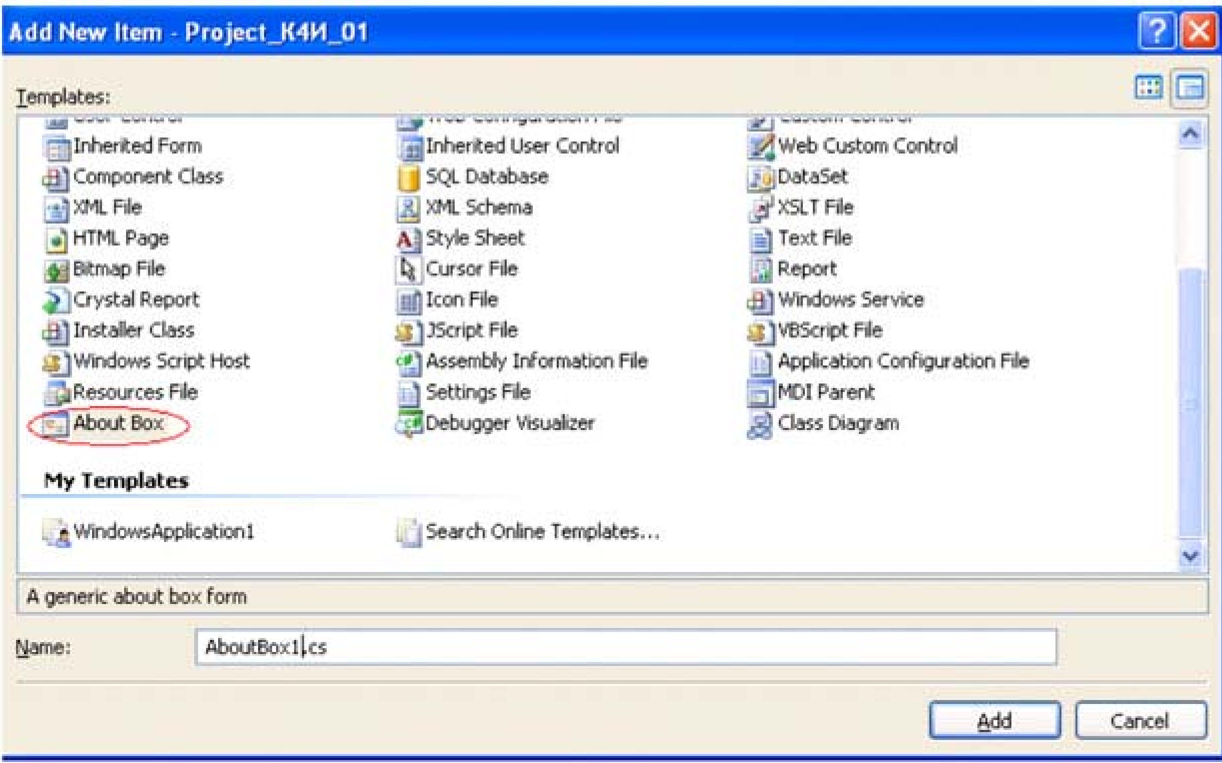


Рис. 4.3. Выбор режима добавления нового компонента в проект



увеличить изображение

Рис. 4.4. Добавление нового компонента в проект

Для класса AboutBox можно задать *логотип* и дополнительную информацию. *По* умолчанию данный *класс* берет дополнительную информацию из метаданных сборки. Проверьте это.

Мы введем собственную информацию. Для этого изменим фрагмент кода конструктора класса AboutBox1 следующим образом.

```
public AboutBox1()
{
  InitializeComponent();
  this.Text = String.Format("О программе {0}", AssemblyTitle);
  this.labelProductName.Text = AssemblyProduct;
  this.labelVersion.Text = String.Format("Version {0}", AssemblyVersion);
  this.labelCopyright.Text = "@РГЭУ, 2008";
  this.labelCompanyName.Text = "Долженко А.И.";
  this.textBoxDescription.Text = "Дисциплина Современные технологии 
программирования. Студенческий проект";
}
```

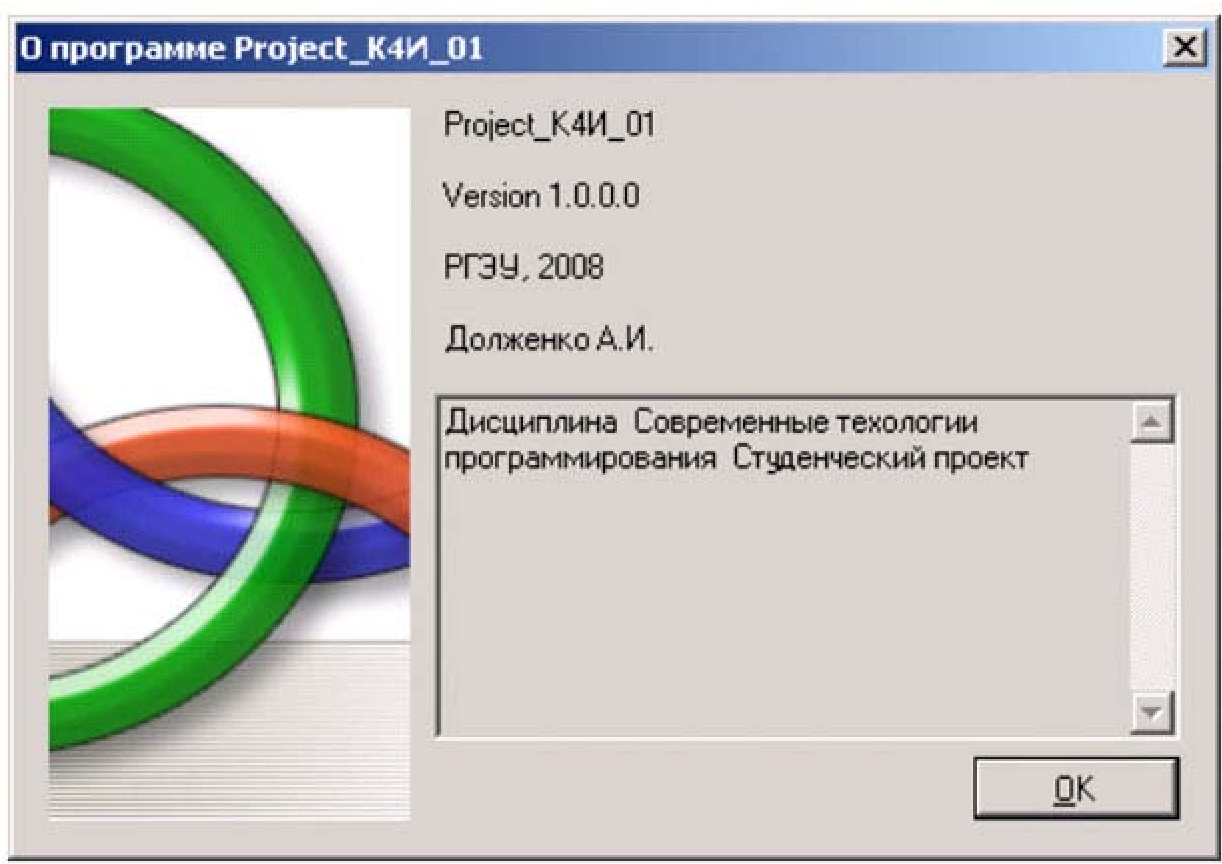
Для открытия пользовательского модального диалогового окна используется метод ShowDialog. В лабораторной работе *диалоговое окно* должно открываться при щелчке пользователем на пункте в *меню* "Справка/О программе". Код для открытия диалогового окна может выглядеть следующим образом:

```
// Открываем модальное диалоговое окно
private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{
```

```
    AboutBox1 aboutBox = new AboutBox1();
    aboutBox.ShowDialog(this);
}
```

Модальность формы определяет именно метод `ShowDialog`: при использовании кода ход выполнения программы будет приостановлен вплоть до того момента, пока метод `ShowDialog` не вернет соответствующее *значение*. Для пользователя это значит, что ему придется закрыть *диалоговое окно*, прежде чем он сможет выполнить какие-либо *операции* на главной форме.

После компиляции и загрузки приложения, вызвав *пункт меню* "Справка/О программе" на дисплеи будет выведено *диалоговое окно*, приведенное на рисунке 4.5.



увеличить изображение

Рис. 4.5. Вызов модального окна

При нажатии на кнопку OK *диалоговое окно* будет автоматически закрыто и в ходе дальнейшего выполнения программы можно выяснить *значение* свойства `DialogResult`.

Задание на практическое занятие

1. Изучить теоретический материал.
2. Создать модальное диалоговое окно с помощью класса `MessageBox`.
3. Создать пользовательское модальное диалоговое окно для пункта меню "О программе".
4. Написать обработчики для вызова модальных окон.
5. Протестировать работу приложения.

Практическое занятие 5. Создание панели инструментов и контекстного меню

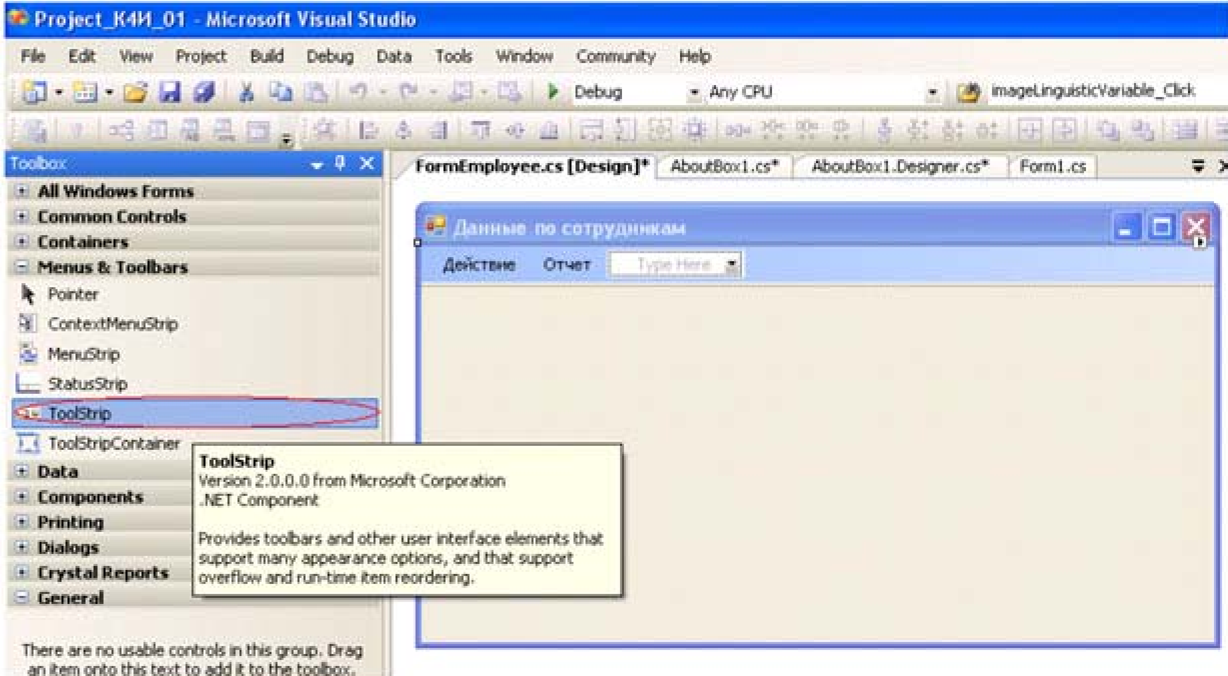
Цель занятия: Изучить основные способы построения панели инструментов и контекстного *меню*, их параметры и получить практические навыки в разработке.

В приложении для повышения качества интерфейса пользователя целесообразно предусматривать различные способы активизации функций системы. При выполнении лабораторной работы 3 для приложения было создано *меню*, которое позволяет активизировать функции "Отменить", "Создать", "Редактировать", "Сохранить" и "Удалить". Данные функции могут быть активизированы с помощью кнопок панели инструментов и контекстного *меню*.

Разработка панели инструментов

Элемент управления ToolStrip используется непосредственно для построения панелей инструментов. Данный элемент использует набор элементов управления, происходящих от класса `ToolStripItem`.

В Visual Studio.NET предусмотрены средства, которые позволяют добавить *панель инструментов* при помощи графических средств. Для этого необходимо открыть панель Toolbox и добавьте элемент управления ToolStrip (рисунок 5.1) на разрабатываемую форму FormEmployee.



увеличить изображение

Рис. 5.1. Окно свойств панели инструментов

В выпадающем *меню* элемента управления ToolStrip на форме FormEmployee необходимо выбрать элемент управления button - кнопка (рисунок 5.2). При этом в панели инструментов добавится кнопка.

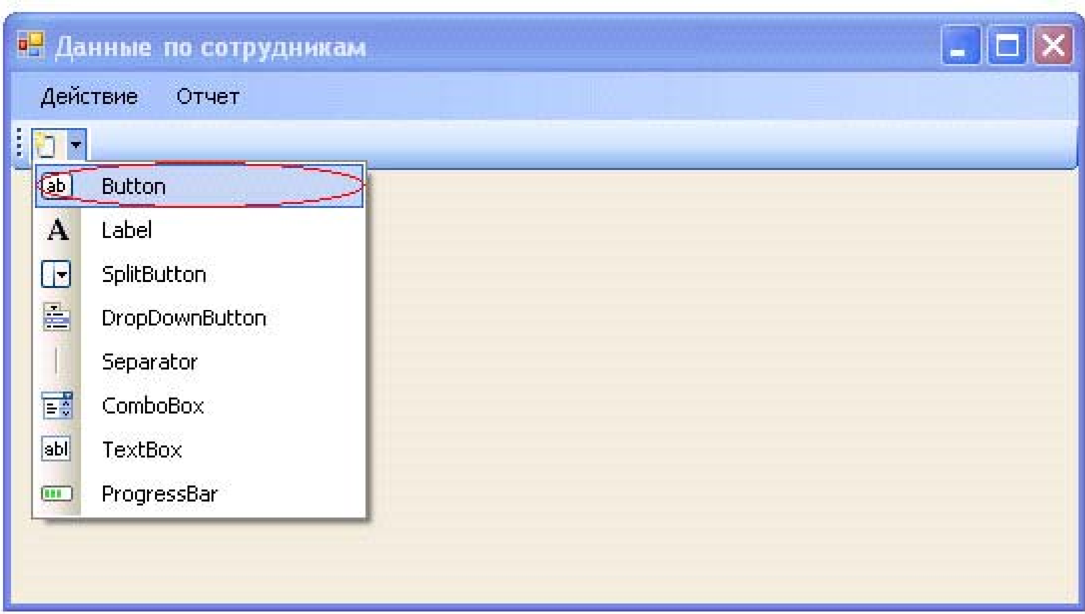


Рис. 5.2. Окно свойств панели инструментов

Добавьте на *панель инструментов* кнопки с именами toolStripButtonUndo, toolStripButtonNew, toolStripButtonEdit, toolStripButtonSave, toolStripButtonRemove. В результате должна быть сформирована *панель инструментов* с кнопками (рисунок 5.3).

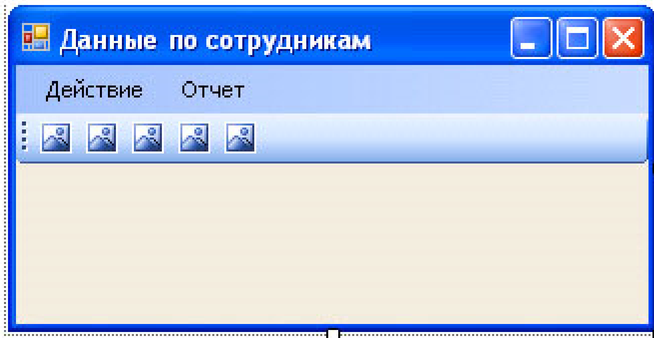


Рис. 5.3. Форма FormEmployee с панелью инструментов

Для кнопок панели инструментов сформируем графическое *представление*. Это можно сделать путем задания свойства Image соответствующей кнопке (рисунок 5.4).

При открытии коллекции свойства Image соответствующей кнопки, нажатии кнопки ... открывается окно мастера выбора графического ресурса (рисунок 5.5).

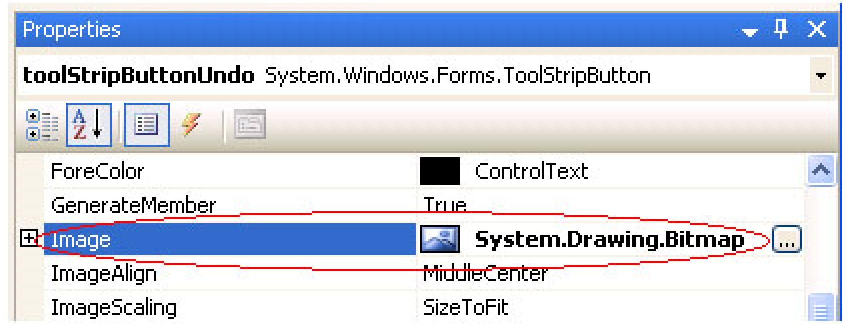


Рис. 5.4. Свойство Image для кнопки панели инструментов

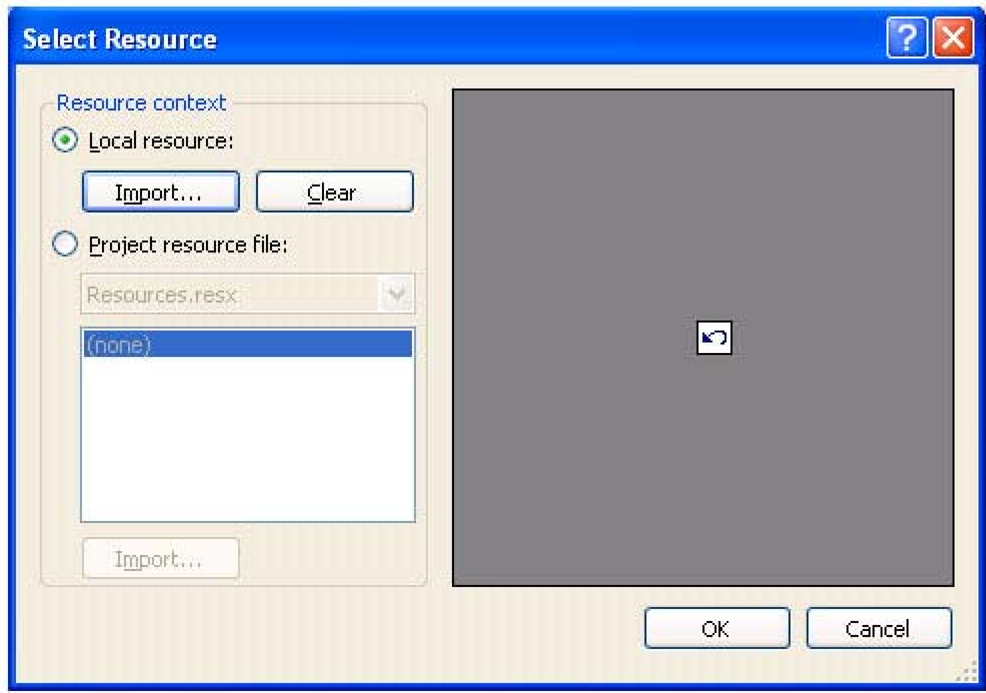


Рис. 5.5. Добавление изображения в ImageList

С помощью кнопки Import в локальный *ресурс* добавляют ссылки на необходимые графические файлы, для формирования изображения кнопок. Результаты формирования графического представления кнопок *панель инструментов* приведены на рисунке 5.6. Графические файлы расположены в папке *Visual Studio*2005\VS2005ImageLibrary\bitmaps\commands\16color (для лабораторной работы графические файлы можно найти в папке Лабораторные работы).

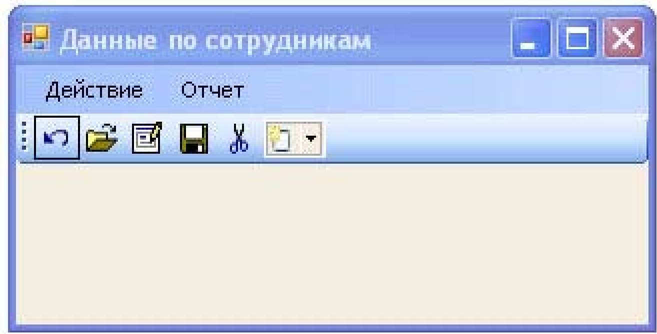


Рис. 5.6. Форма с панелью инструментов

Каждая кнопка панели инструментов, которая является объектом класса `toolStripButton`, может содержать текст, или изображение, или и то и другое.

Созданная *панель инструментов* содержит пять кнопок. *По* функциональности каждой из этих кнопок будут соответствовать пункты *меню* "Отменить", "Создать", "Редактировать", "Сохранить" и "Удалить".

Для удобства пользователя целесообразно снабдить кнопки панели инструментов всплывающими подсказками при фокусировке курсора на данной кнопке. Это можно сделать, если свойству `ToolTipText` класса `toolStripButton` задать *значение* текстовой строки с содержанием подсказки. На рисунке 5.7 для кнопки "Отменить" (`toolStripButtonUndo`) строка подсказки `ToolTipText` соответствует строке "Отменить".

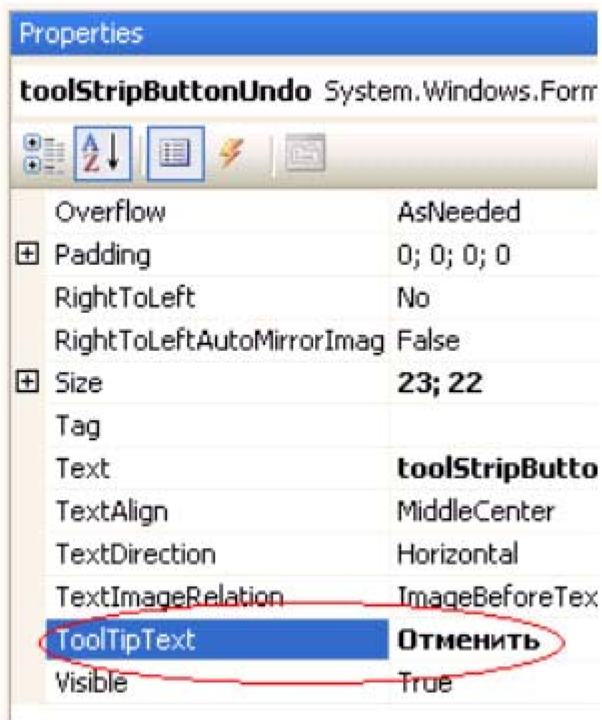


Рис. 5.7. Формирование подсказки для кнопки

На рисунке 5.8 показан *вывод* подсказки при фокусировке курсора на кнопке панели инструментов.

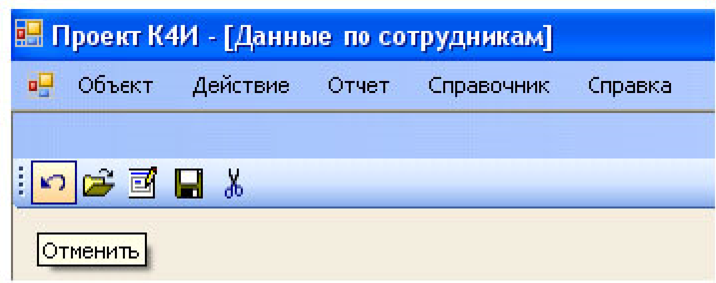


Рис. 5.8. Вывод подсказки для кнопки

Для распознавания реакции приложения при нажатии кнопок панели инструментов необходимо создать обработчик события для кнопок. При двойном нажатии на кнопку панели инструментов генерируется обработчик, в который нужно добавить *вызов метода* `Undo`. В этом случае обработчик нажатия кнопки панели инструментов будет иметь следующий вид:

```
private void toolStripButtonUndo_Click(object sender, EventArgs e)
{
    Undo();
}
```

Контекстное меню

Класс ContextMenuStrip применяется для показа контекстного *меню*, или *меню*, отображаемого *по* нажатию правой кнопки мыши. Для создания объекта класса ContextMenuStrip необходимо открыть панель Toolbox и добавить элемент управления contextMenuStrip на форму FormEmployee (рисунок 5.9).

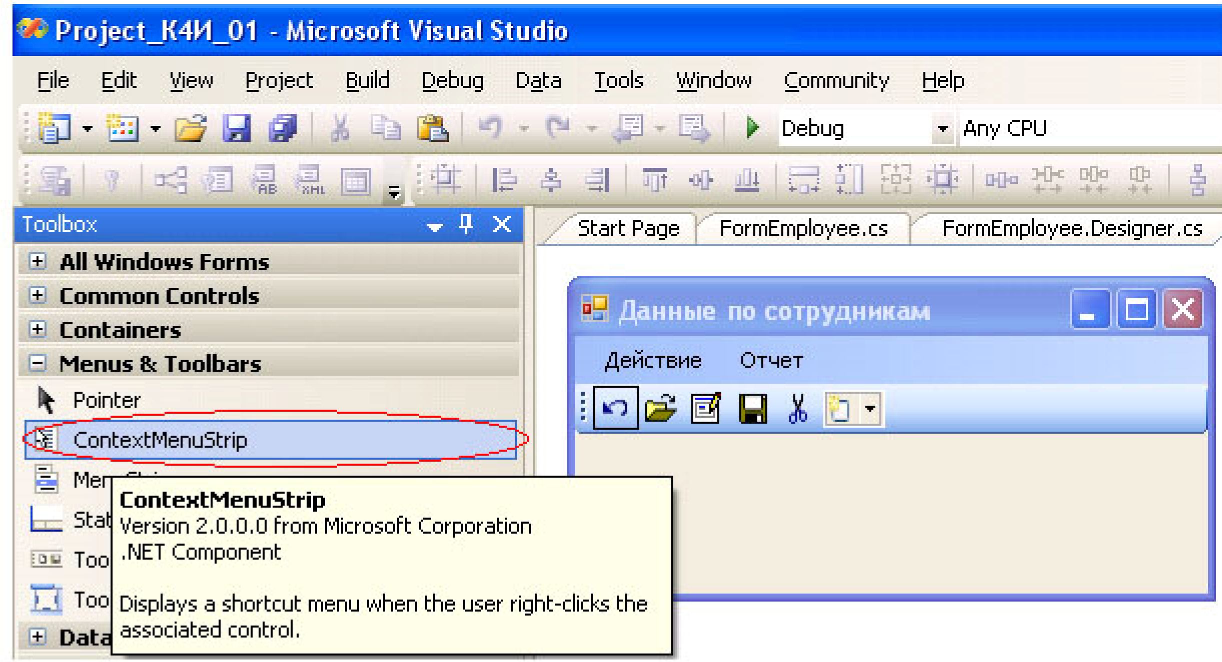


Рис. 5.9. Формирование на форме контекстного меню

В результате получаем форму FormEmployee с контекстным *меню* (рисунок 5.10)

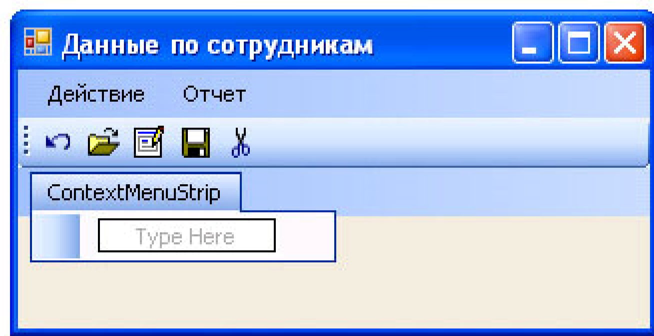


Рис. 5.10. Форма с контекстным меню

Формирование пунктов контекстного *меню* производится аналогично формированию пунктов главного *меню* (смотри лабораторную работу 2). Сформированное контекстное *меню* приведено на рисунке 5.11.

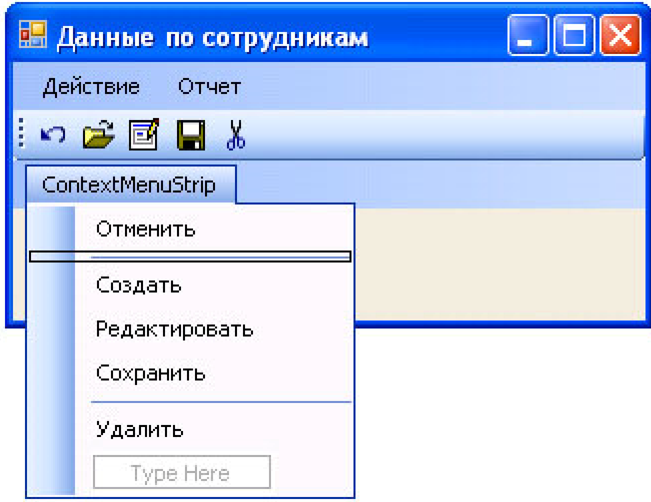


Рис. 5.11. Вид контекстного меню

После формирования пунктов контекстного *меню* необходимо его подключить к форме FormEmployee. Для этого на вкладке Свойства (Properties) формы FormEmployee строке, соответствующей свойству `ContextMenuStrip` нужно установить *значение* созданного объекта `contextMenuStrip1` (рис. 5.12)

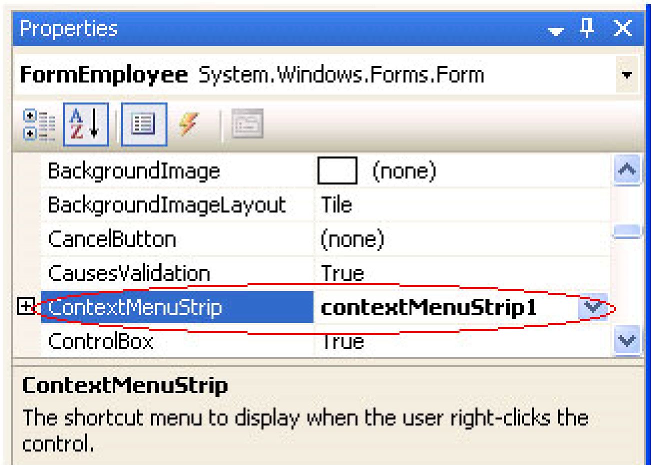


Рис. 5.12. Подключение контекстного меню к форме

После компиляции проекта и запуска приложения на выполнение можно проверить режим активизации контекстного *меню*. Для этого необходимо выбрать из главного *меню пункт* "Сотрудник" и на появившейся форме в любом месте щелкнуть правой кнопкой мыши. Результат всплытия на форме контекстного *меню* показан на рисунке 5.13.

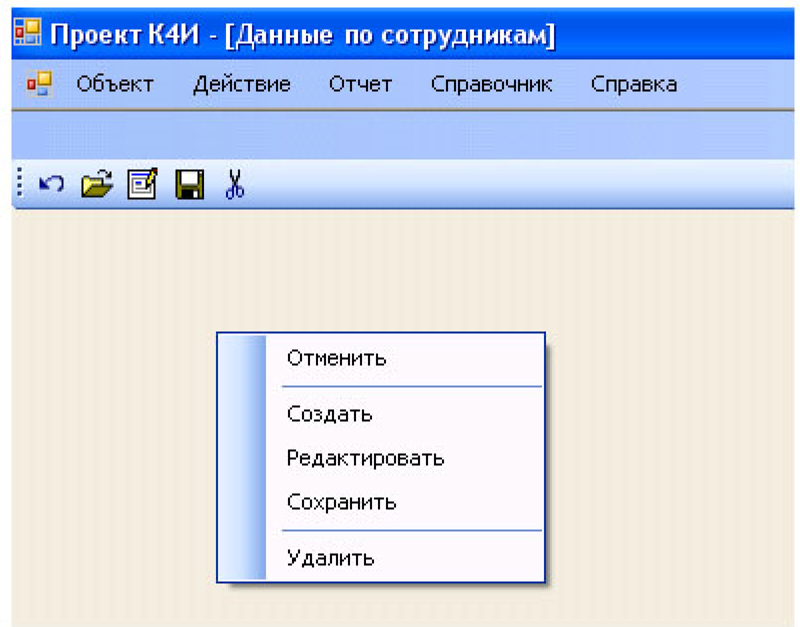


Рис. 5.13. Активизация контекстного меню

Привязка пунктов контекстного *меню* к конкретным функциям осуществляется путем создания кода обработчика событий для каждого пункта *меню*. Для формирования обработчика необходимо перейти в окно дизайнера формы FormEmployee, выделить на форме *класс* `ContextMenuStrip` и сделать *двойной щелчок* на соответствующем пункте *меню*, например "Отменить". В сгенерированном обработчике необходимо добавить *вызов метода*, для функции "Отменить" - метод `Undo( )`. Листинг обработчика метода приведен ниже.

```
private void undo1ToolStripMenuItem_Click(object sender, EventArgs e)
{
    Undo();
}
```

Задание на практическое занятие

1. Изучить теоретический материал.
2. Создать панель инструментов.
3. Создать контекстное меню.
4. Написать обработчики для панели инструментов и контекстного меню.
5. Протестировать работу приложения

Практическое занятие 6. Создание строки состояния

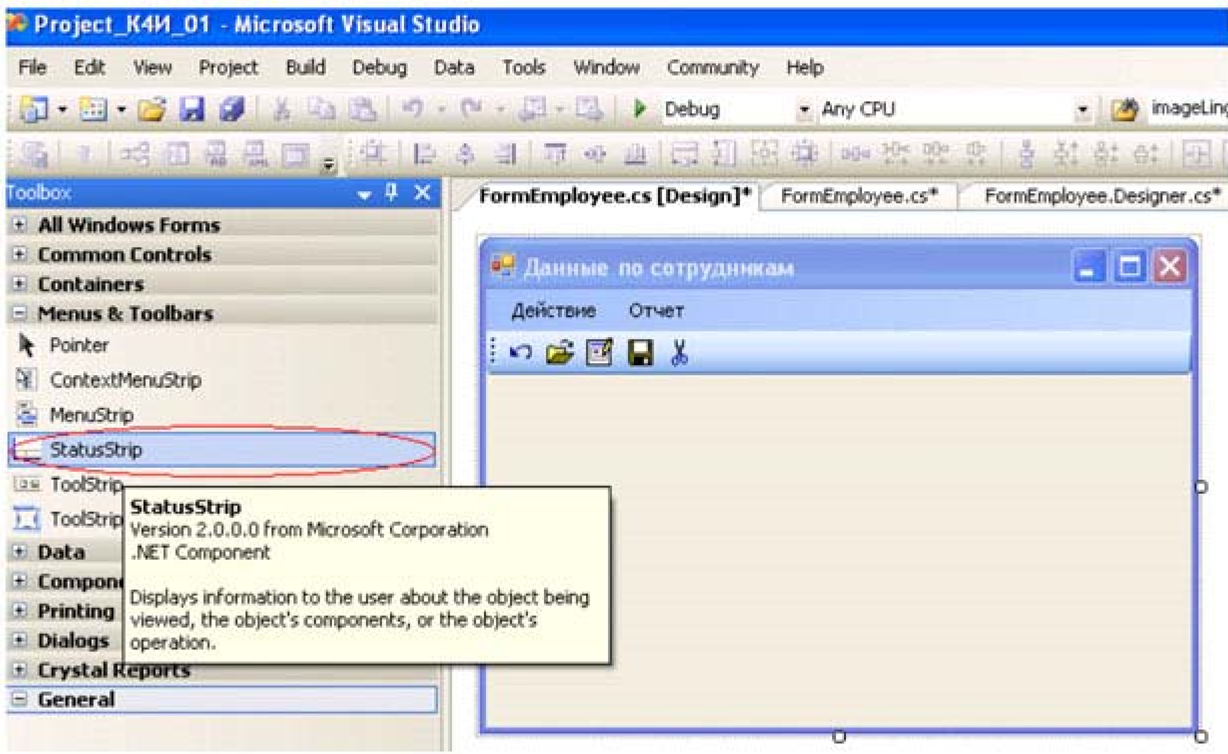
Цель занятия: Изучить основные способы построения строки состояния и получить практические навыки в разработке

Создание строки состояния

На многих формах в реальных приложениях имеется элемент интерфейса, называемый строкой состояния (StatusStrip). Обычно в строке состояния выводится некоторая текстовая или графическая *информация*, относящаяся к работе приложения. Строка состояния может быть разделена на несколько "панелей" (*panel*) - отдельных частей окна. В каждой из этих панелей *информация* выводится отдельно.

Создадим строку состояния, в которой будут выводиться текстовые сообщения, относящиеся к пунктам *меню*.

В окне *Toolbox* выделим *пункт StatusStrip* и перетащим его на форму (рисунок 6.1).



увеличить изображение

Рис. 6.1. Добавляем на форму строку состояния

Объекту класса StatusStrip присвоим имя statusStripEmployee. Откроем выпадающий *список* объекта класса statusBarEmployee и выберем *объект* StatusLable (рисунок 6.2). Присвоим ему имя toolStripStatusLabelEmployee.

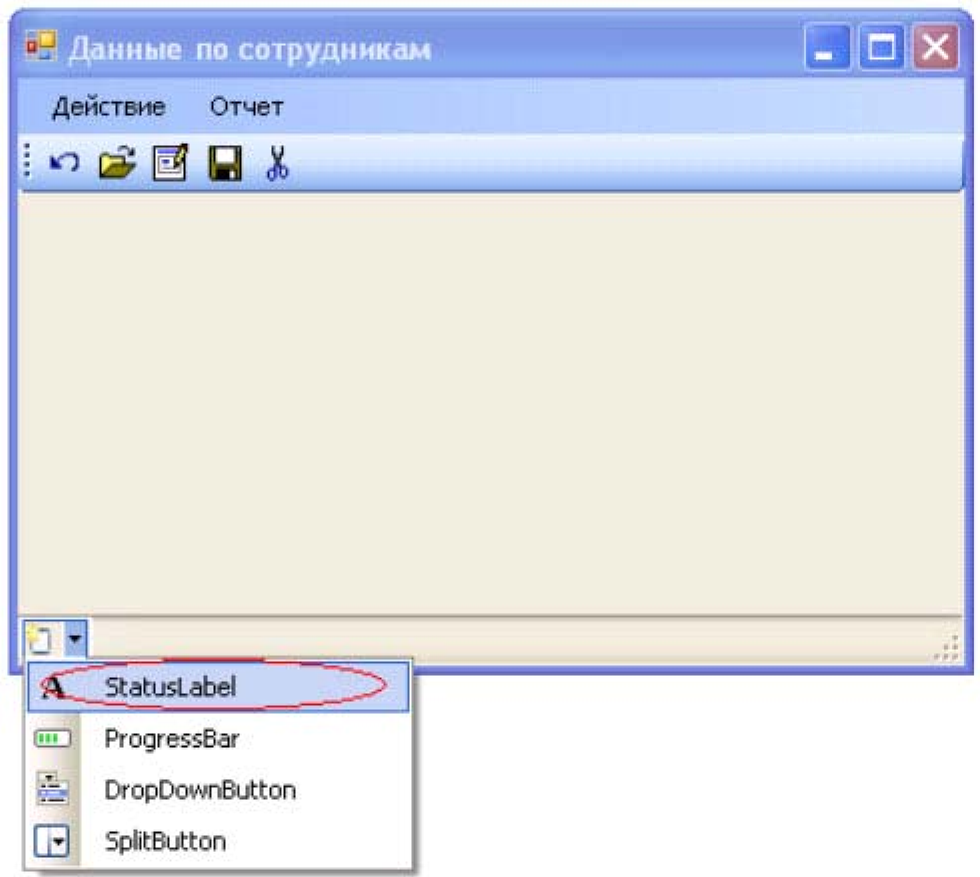


Рис. 6.2. Добавляем метку в строку состояния

При компиляции, запуске приложения и выборе пункта *меню* "Сотрудник" *экранная форма* будет иметь вид, представленный на рисунке 6.3.

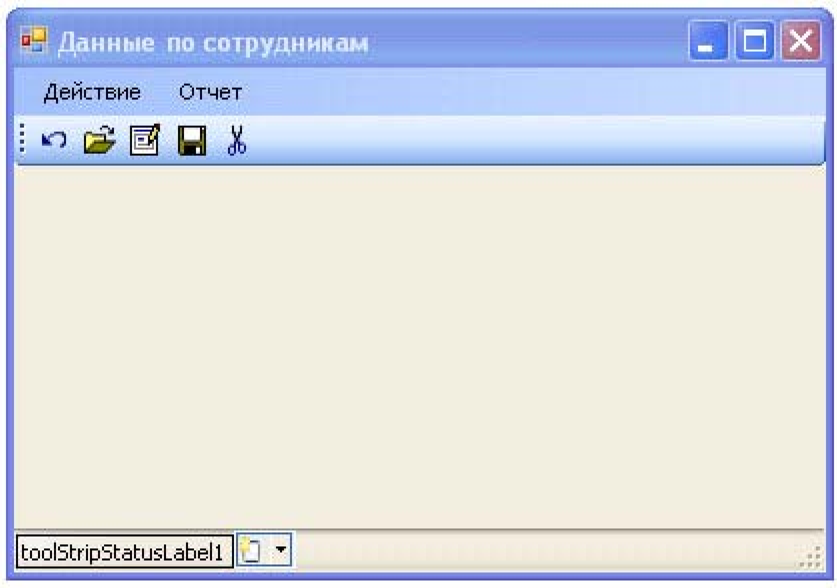


Рис. 6.3. Окно приложения со строкой состояния

Для управления текстом строки состояния необходимо разработать обработчик события для соответствующих объектов.

Для формы *FormEmployee* в строке состояний необходимо вывести информацию при наведении курсора мыши на пунктах *меню "Действие"*. Первоначально в дизайнере формы необходимо

выделить *пункт меню "Действие"*, перейти на вкладку *Properties* и открыть окно событий, нажав кнопку . На данной вкладке необходимо выделить событие `MouseEnter` и в *поле* ввода сделать *двойной щелчок*. (рисунок 6.4).

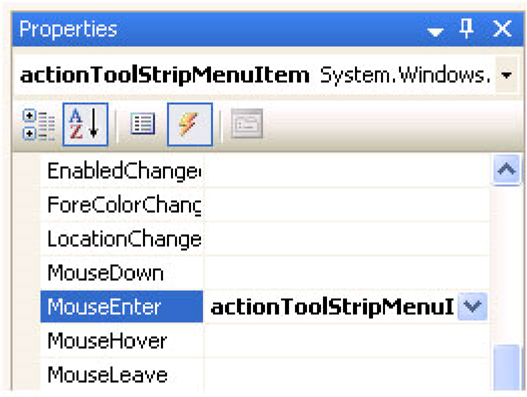


Рис. 6.4. Окно событий

Система сгенерирует код обработчика, приведенного ниже.

```
private void actionToolStripMenuItem_MouseEnter(object sender, EventArgs e)
{    }
```

Добавим в обработчик следующий код:

```
private void actionToolStripMenuItem_MouseEnter(object sender, EventArgs e)
{
   toolStripStatusLabelEmployee.Text =
        "Выбор действий по сотрудникам";
}
```

Если откомпилировать программу, запустить её, выбрать *пункт меню "Сотрудник"* и навести *указатель* мыши на *пункт "Действие"*, то сгенерируется событие `"MouseEnter"` и в строке состояния выведется текстовое сообщение `"Выбор действий по сотрудникам"` (рисунок 6.5).

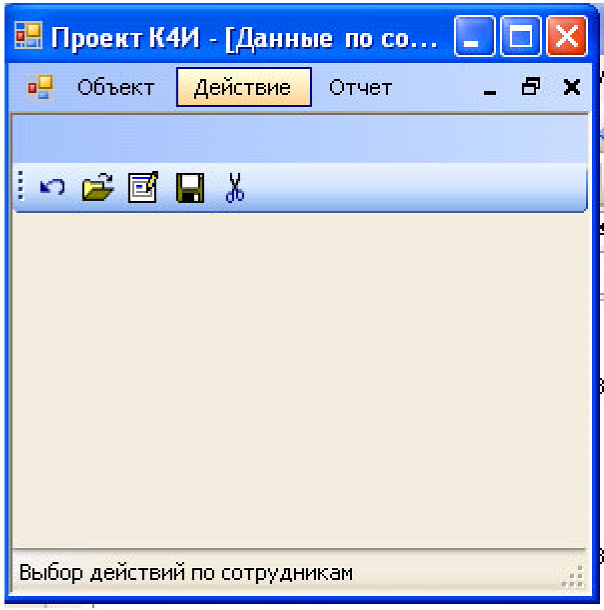


Рис. 6.5. Вывод сообщения в строке состояния

Если теперь переместить *указатель* мыши с пункта *меню "Действие"*, то текст в строке состояния не изменится. Такой режим работы программы является неправильным, так как если *указатель* мыши перемещается с пункта *меню "Действие"*, то строка состояния должна становиться пустой. Для обеспечения правильной работы программы воспользуемся ещё одним событием `"MouseLeave"`, которое генерируется, когда *мышь* перемещается (покидает) с пункта *меню "Действие"*. Обработчик данного события имеет следующий вид:

```
private void actionToolStripMenuItem_MouseLeave(object sender, EventArgs e)
{
    toolStripStatusLabelEmployee.Text = "";
}
```

Вышеприведенные обработчики будут вызываться только тогда, когда *пользователь* наведет *указатель* мыши на *пункт меню "Действие"*. Для того чтобы обработчики реагировали на все строки пунктов главного *меню "Действие"* и *"Отчет"* формы *FormEmployee* необходимо сформировать соответствующие события`MouseEnter` и `MouseLeave` для всех подпунктов *меню* и создать для них обработчики.

Результаты компиляции и выполнения приложения приведены на рисунке 6.6.

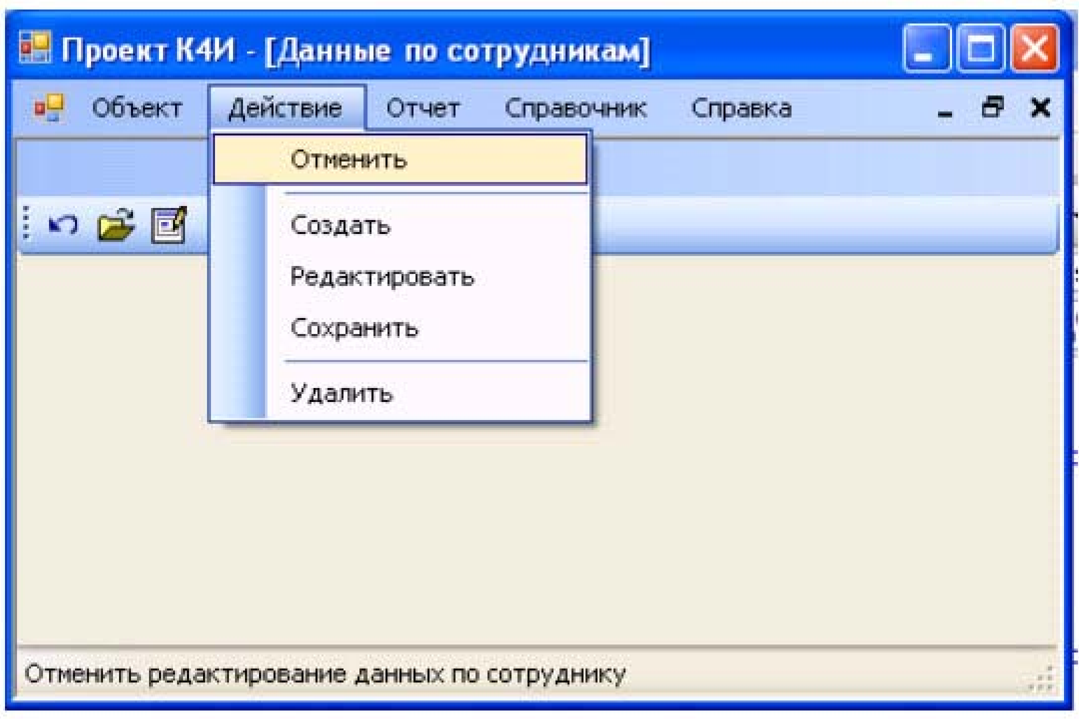


Рис. 6.6. Экранная форма со строкой состояния

Задание на практическое занятие

1. Изучить теоретический материал.
2. Создать строку состояний для главной и дочерней форм.
3. Написать обработчики для формирования строки состояний, отображающих информацию о пунктах меню.
4. Протестировать работу приложения.

Практическое занятие 7. Создание элементов управления

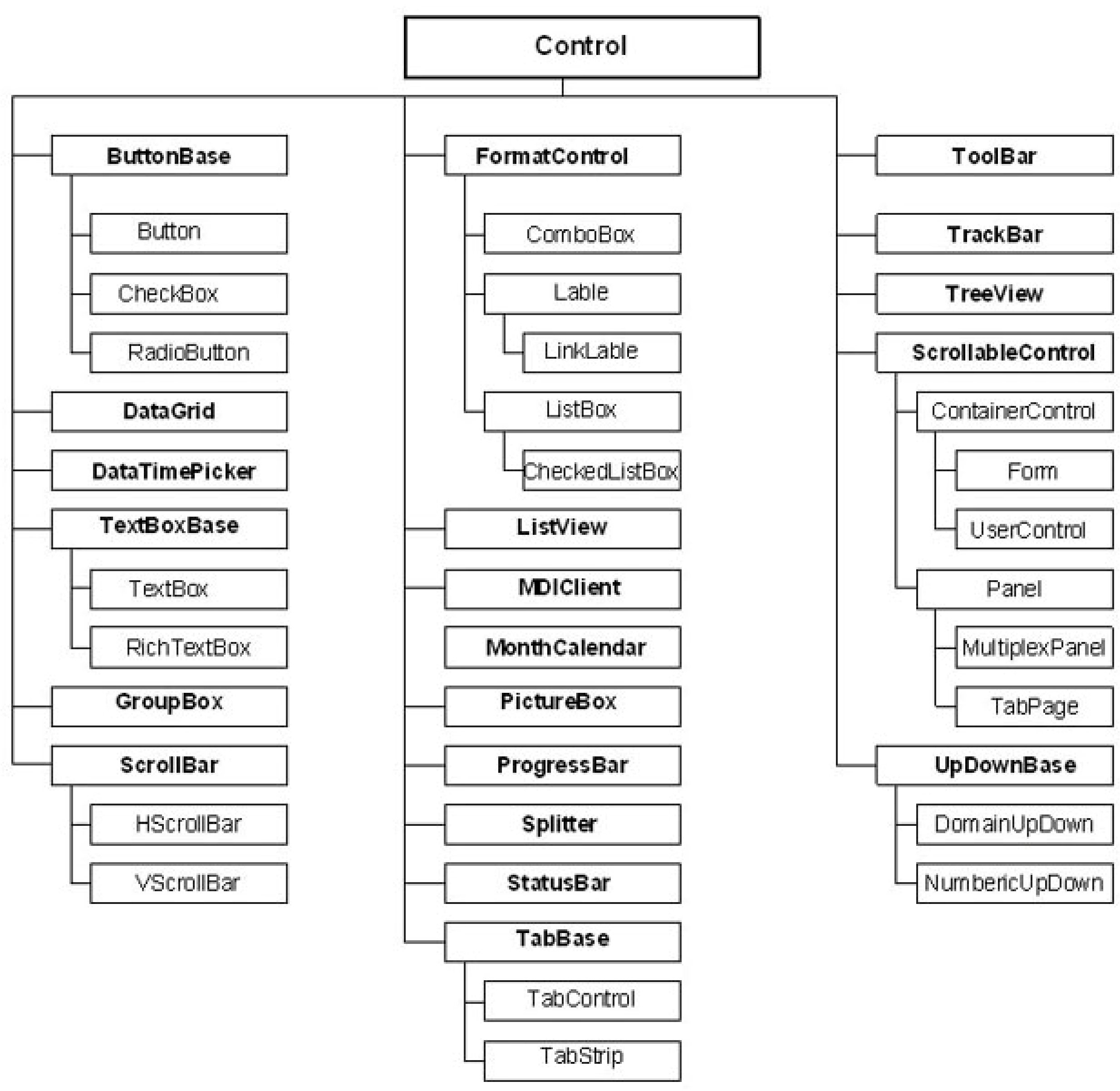
Цель занятия: Изучить основные *элементы управления Windows* -форм, их свойства и методы, а также получить практические навыки в разработке *Windows* -форм с элементами контроля

Общие сведения

Классы, представляющие графические *элементы управления*, находятся в пространстве имен *System.Windows.Forms*. С их помощью обеспечивается *реакция* на действия пользователя в приложении *Windows Forms*. Классы элементов управления связаны между собой достаточно сложными отношениями наследования. Общая схема таких отношений представлена на рисунке 7.1.

Класс `Control`, как общий предок, обеспечивает все *производные* классы общим набором важнейших возможностей. В числе этих возможностей можно перечислить события мыши и клавиатуры, физические размеры и местонахождение элемента управления (свойства `Height, Width, Left, Right, Location`), установку цвета фона и цвета переднего плана, выбор шрифта и т.п.

При создании приложения можно добавить *элементы управления* на форму при помощи графических средств *Visual Studio*. Обычно достаточно выбрать нужный элемент управления в окне *ToolBox* и поместить его на форме. *Visual Studio* автоматически сгенерирует нужный код для формы. После этого можно изменить название элемента управления на более содержательное (например, вместо `button1`, предлагаемого по умолчанию, - `buttonPrimer`) *Visual Studio* позволяет не только размещать на форме *элементы управления*, но и настраивать их свойства. Для этого достаточно щелкнуть на элементе управления правой кнопкой мыши и в контекстном *меню* выбрать *Properties* (Свойства).



увеличить изображение

Рис. 7.1. Иерархия элементов управления Windows Forms

Все изменения, которые необходимо произвести в открывшемся окне (рисунок 7.2), будут добавлены в код метода `InitializeComponents()`.

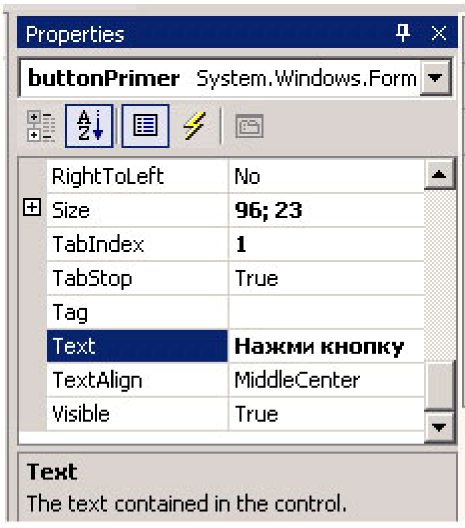


Рис. 7.2. Настройка элементов управления средствами Visual Studio

То же самое окно позволяет настроить не только свойства данного элемента управления, но и обработку событий этого элемента. Перейти в *список событий* можно при помощи кнопки в закладке *Properties* (рисунок 7.3). Можно выбрать в списке нужное событие и рядом с ним сделать *двойной щелчок* или ввести имя метода или выбрать метод из списка.

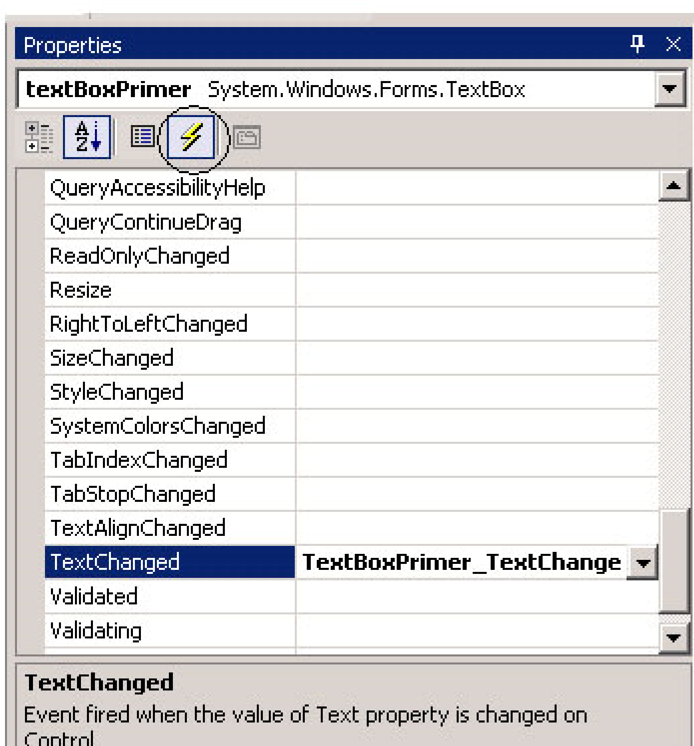


Рис. 7.3. Настройка обработчиков событий

После задания имени метода или двойного щелчка *Visual Studio* сгенерирует заготовку для обработчика события.

Рассмотрим основные *элементы управления Windows* -форм.

Элемент управления TextBox

Элемент управления TextBox (текстовое окно) предназначен для хранения текста (одной или нескольких строк). При желании текст в TextBox может быть настроен как "только для чтения", а в правой и нижней части можно поместить полосы прокрутки.

Класс TextBox происходит непосредственно от класса TextBoxBase, обеспечивает общими возможностями как TextBox, так и RichTextBox. Свойства, определенные в TextBoxBase, представлены в таблице 7.1.

Таблица 7.1. Свойства TextBoxBase

Свойство	Назначение
AcceptsTab	Определяет, что будет производиться при нажатии на клавишу *Tab*: вставка символа табуляции в само поле или переход к другому элементу управления
AutoSize	Определяет, будет ли элемент управления автоматически изменять размер при изменении шрифта на нем
BackColor, ForeColor	Позволяют получить или установить значение цвета фона и переднего плана
HideSelection	Позволяет получить или установить значение, определяющее, будет ли текст в TextBox оставаться выделенным после того, как этот элемент управления будет выведен из фокуса
MaxLength	Определяет максимальное количество символов, которое можно будет ввести в TextBox
Modified	Позволяет получить или установить значение, определяющее, был ли текст в TextBox изменен пользователем
Multiline	Указывает, может ли TextBox содержать несколько строк текста
Readonly	Помечает TextBox как "только для чтения"
SelectedText, SelectionLength	Содержат выделенный текст (или определенное количество символов) в TextBox

SelectionStart	Позволяет получить начало выделенного текста в TextBox
Wordwrap	Определяет, будет ли текст в TextBox автоматически переноситься на новую строку при достижении предельной длины строки

В TextBoxBase также определено множество методов: для работы с буфером обмена (Cut, Copy и Paste), отменой ввода (Undo) и прочими возможностями редактирования (Clear, AppendText и т. п.).

Из всех событий, определенных в TextBoxBase, наибольший интерес представляет событие TextChange. Это событие происходит при изменении текста в объекте класса, производном от TextBoxBase. Например, его можно использовать для *проверки допустимости* вводимых пользователем символов (например, предположим, что пользователь должен вводить в поле только цифры или, наоборот, только буквы).

Свойства, унаследованные от Control и от TextBoxBase, определяют большую часть возможностей TextBox. Свойств, определенных непосредственно в классе TextBox, не так уж и много. Они представлены в таблице 7.2.

Таблица 7.2. Свойства, определенные в классе TextBox

Свойство	**Назначение**
AcceptsReturn	Позволяет определить, что происходит, когда пользователь при вводе текста нажал на Enter. Варианта два: либо в TextBox начинается новая строка текста, либо активизируется кнопка по умолчанию на форме
CharacterCasing	Позволяет получить или установить значение, определяющее, будет ли изменяться регистр вводимых пользователем символов
PasswordChar	Позволяет выбрать символ, используемый для отображения вводимых пользователем данных (в поле для ввода пароля)
ScrollBars	Позволяет получить или установить значение, определяющее, будут ли в TextBox с несколькими строками присутствовать полосы прокрутки
TextAlign	Позволяет определить выравнивание текста в TextBox (используются значения из перечисления HorizontalAlignment)

Значения перечисления HorizontalAlignment представлены в таблице 7.3.

Таблица 7.3. Значения перечисления HorizontalAlignment	
Значение	**Описание**
Center	Выравнивание по центру
Left	Выравнивание по левому краю
Right	Выравнивание по правому краю

Класс Button

Кнопка (button) - это самый простой из всех элементов управления и при этом наиболее часто используемый. Можно сказать, что кнопка - это возможность принять ввод (щелчок кнопкой мыши или набор на клавиатуре) наиболее простым способом. Непосредственный предок класса System.Windows.FormButton в иерархии классов *.NET* - это класс ButtonBase, обеспечивающий общие возможности для целой группы производных от него элементов управления (таких как Button, CheckBox и RadioButton). Некоторые свойства ButtonBase представлены в таблице 7.4.

Таблица 7.4. Свойства ButtonBase	
Свойство	**Назначение**
FlatStyle	Позволяет настроить "рельефность" кнопки. Используются значения из перечисления FlatStyle
Image	Позволяет задать изображение, которое будет выводиться на кнопке (при этом можно указать точное местонахождение изображения). Фоновый рисунок лучше настраивать при помощи свойства BackgroundImage, определенного в базовом классе Control
ImageAlign	Позволяет определить выравнивание изображения, размещенного на кнопке. Используются значения из перечисления ContentAlignment
ImageIndex, ImageList	Эти свойства используются для работы с набором изображений (объектом ImageList), выводимых на кнопке
IsDefault	Определяет, будет ли эта кнопка являться кнопкой по умолчанию (то есть срабатывать при нажатии на *Enter*)

TextAlign	Позволяет получить или установить выравнивание текста на кнопке. Также используются значения из перечисления ContentAlignment

Сам класс Button не определяет каких-либо дополнительных возможностей помимо унаследованных от ButtonBase, за единственным, но существенным исключением свойства DialogResult. Это свойство позволяет возвращать значение при закрытии диалогового окна, например, при нажатии кнопок *OK* или *Cancel*(Отменить).

В подавляющем большинстве случаев выравнивание текста, размещенного на кнопке, производится по центру, так что текст будет размещен строго посередине кнопки. Однако если нам по каким-то причинам необходимо использовать другой стиль выравнивания, в нашем распоряжении - свойство TextAlign, определенное в классе ButtonBase. Для TextAlign используются значения из перечисления ContentAlignment (таблица 7.5). Значения из того же перечисления используются и для определения положения изображения на кнопке.

Таблица 7.5. Значения перечисления ContentAlignment

Значение	**Описание (выравнивание)**
BottomCenter	По нижнему краю кнопки, относительно боковых краев - посередине
BottomLeft	Но нижнему краю кнопки, слева
BottomRight	По нижнему краю кнопки, справа
MiddleCenter	По центру кнопки
MiddleLeft	Относительно верхнего и нижнего краев - по центру, относительно боковых краев - слева
MiddleRight	Относительно верхнего и нижнего краев - по центру, относительно боковых краев - справа
TopCenter	По верхнему краю кнопки, относительно боковых краев - посередине
TopLeft	По верхнему краю кнопки, слева
TopRight	По верхнему краю кнопки, справа

Флажки

Для флажка (тип `CheckBox`) предусмотрено три возможных состояния. Как и тип `Button`, `CheckBox` наследует большую часть своих возможностей от базовых классов `Control` и `ButtonBase`. Однако в этом классе существуют и свои собственные члены, обеспечивающие дополнительные уникальные возможности. Наиболее важные свойства `CheckBox` представлены в таблице 7.6.

Таблица 7.6. Свойства класса CheckBox	
Свойство	**Назначение**
`Appearance`	Настраивает вид флажка. Для этого свойства используются значения из перечисления `Appearance`
`AutoCheck`	Позволяет получить или установить значение, определяющее, будут ли значения Checked и CheckState, а также внешний вид флажка автоматически изменяться при щелчке на нем
`CheckAlign`	Позволяет установить горизонтальное и вертикальное выравнивание собственно флажка (квадратика) в элементе управления `CheckBox`. Используются значения из перечисления `ContentAlignment`
`Checked`	Возвращает значение типа `bool`, представляющее текущее состояние флажка (выбран или не выбран) Если для свойства `ThreeState` установлено значение `true`, то свойство `Checked` будет возвращать `true` как для явно выбранного флажка, так и для того флажка, для которого установлено значение "не определено" (*`indeterminate`*)
`CheckState`	Позволяет получить или установить значение флажка (установлен - не установлен - не определено), используя не `true` и `false`, как в `Checked`, а три значения из перечисления `CheckState`. Обычно используется, если свойство`ThreeState` для флажка имеет значение `true` (то есть он допускает три значения).
`ThreeState`	Определяет, будут ли для флажка использоваться три значения (из перечисления `CheckState`) или только два

Возможные состояния флажка (*`Indeterminate`* можно использовать только тогда, когда для свойства `ThreeState` установлено значение `true`) представлены в таблице 7.7.

Таблица 7.7. Значения перечисления CheckState

Значение	Описание
Checked	Флажок установлен
Indeterminate	Значение не определено (обычно флажок выглядит как "серый", затененный)
Unchecked	Флажок снят

Состояние "значение не определено" (*indeterminate*) может быть установлено, например, для верхнего элемента иерархии, в которой для одной части подчиненных элементов флажок установлен, а для другой - снят.

Переключатели и группирующие рамки

Тип RadioButton (переключатель) можно воспринимать, как несколько видоизмененный флажок при этом сходство между этими типами подчеркивается почти полным совпадением наборов членов. Между типами RadioButton и CheckBox существуют лишь два важных различия: в RadioButton предусмотрено событие CheckedChanged (возникающее при изменении значения Checked), а кроме того, RadioButton не поддерживает свойство ThreeState и не может принимать состояние *Indeterminate* (не определено).

Переключатели всегда используются в группах, которые рассматриваются как некое единое целое. Внутри группы переключателей одновременно может быть выбран только один переключатель. Для группировки переключателей в группы используется тип GroupBox.

И флажок (CheckBox), и переключатель (RadioButton) поддерживают свойство Checked, при помощи которого очень удобно получать информацию о состоянии соответственно флажка и переключателя. Однако если есть необходимость задействовать дополнительное третье состояние флажка (не определено - *Indeterminate*), то придется вместо Checked использовать свойство CheckState и значения из одноименного перечисления CheckState.

Элемент управления CheckedListBox

Типы Button, CheckBox и RadioButton являются производными от ButtonBase, и их можно определить как некие разновидности кнопок. К членам семейства списков относятся CheckedListBox (список с флажками), ListBox (список) и ComboBox (комбинированный список).

Элемент управления CheckedListBox (список с флажками) позволяет помещать обычные флажки внутри поля с полосами прокрутки.

Кроме того, в элементе управления CheckedListBox предусмотрена возможность использования нескольких столбцов. Для этого достаточно установить значение true для свойства MultiColumn.

CheckedListBox наследует большинство своих возможностей от типа ListBox. То же самое справедливо и в отношении класса ComboBox. Наиболее важные свойства System.Windows.Forms.ListBox представлены в таблице 7.8.

Таблица 7.8. Свойства класса ListBox

Свойство	Назначение
ScrollAlwaysVisible	Определяет, будет ли полоса прокрутки выводиться всегда
SelectedIndex	Индекс выделенного в настоящий момент элемента в списке (если такой имеется). Если ни один элемент не выделен, то возвращается значение -1
SelectedIndices	Набор индексов выделенных в настоящий момент элементов в списке. Если не выделен ни один элемент, то возвращается пустой набор
SelectedItem	Значение выделенного в настоящий момент элемента. Если ни один из элементов не выделен, то возвращается null
SelectedItems	Возвращает коллекцию значений выделенных элементов (для списков, в которых допускается выбор нескольких значений)
SelectionMode	Определяет число элементов, которые возможно выбрать в списке одновременно. Для этого свойства используются значения из перечисления SelectionMode
Sorted	Определяет, будут ли элементы в списке упорядочены (по алфавиту) или нет
TopIndex	Возвращает индекс первого видимого элемента в списке

Помимо свойств в классе ListBox определены также многочисленные методы. Подавляющее большинство этих методов дублирует возможности, предоставляемые в наше распоряжение свойствами, поэтому мы их рассматривать не будем.

Комбинированные списки

Как и списки (объекты ListBox), комбинированные списки (объекты ComboBox) позволяют пользователю производить выбор из списка заранее определенных элементов. Однако у комбинированных списков есть одно существенное отличие от обычных: пользователь может не только выбрать готовое значение из списка, но и ввести свое собственное. Класс ComboBox наследует большинство своих возможностей от класса ListBox (который, в свою очередь, является производным от Control), однако в нем предусмотрены и собственные важные свойства, представленные в таблице 7.9.

Таблица 7.9. Свойства ComboBox

Свойство	Назначение
DroppedDown	"Раскрывающийся вниз": определяет, будет ли список ниспадающим
MaxDropDownltems	Определяет максимальное количество элементов, которое будет показано в нижней части ниспадающего списка. Допустимые значения - от 1 до 100
MaxLength	Определяет максимальную длину текста, который пользователь может ввести в ComboBox
Selectedlndex	Определяет индекс выделенного элемента ComboBox. Если ни один элемент не выделен, возвращается значение -1
Selectedltem	Возвращает ссылку на объект выделенного элемента ComboBox
SelectedText	Возвращает выделенный текст в поле редактирования ComboBox
SelectionLength	Определяет длину (в символах) выделенного текста в поле редактирования ComboBox
Style	Позволяет получить или установить стиль ComboBox. Для этого свойства используются значения из перечисления ComboBoxStyle
Text	Позволяет получить доступ к тексту в поле редактирования. При работе с ComboBox это унаследованное свойство используется чаще всех остальных

Стиль для ComboBox можно настроить при помощи свойства Style, для которого используются значения из перечисления ComboBoxStyle (таблица 7.10).

Таблица 7.10. Значения перечисления ComboBoxStyle

Значение	Описание

DropDown	Пользователь может вводить значения в поле редактирования. Для отображения списка пользователь должен нажать на кнопку со стрелкой, направленной вниз (Arrow Button)
DropDownList	Пользователь не может вводить значения в поле редактирования. Для отображения списка пользователь должен нажать на кнопку со стрелкой, направленной вниз (Arrow Button)
Simple	Пользователь может вводить значения в поле редактирования. Список значений виден всегда

Порядок перехода по Tab

Если на форме размещено несколько элементов управления, то пользователи обычно ожидают, что между ними можно будет перемещаться с помощью клавиши *Tab*. Часто бывает необходимо после размещения элементов управления настроить порядок перехода между ними. Для этого используются два свойства (унаследованные от базового класса Control и поэтому общие для всех элементов управления): TabStop и Tablndex. Для свойства TabStop используются только два значения: true и false. Если для TabStop установлено значение true, то к этому элементу управления можно будет добраться с помощью клавиши *Tab*. Если же установлено значение false, то участвовать в переходах по *Tab* этот элемент управления не будет. Если элемент управления TabStop имеет значение true, то очередность перехода можно настроить с помощью свойства Tablndex:

В *Visual Studio.NET* предусмотрено средство, при помощи которого можно быстро настроить порядок перехода для элементов управления на форме. Это средство называется *Tab Order Wizard* и оно доступно из меню *View (View > Tab Order)*. Чтобы изменить значения TabIndex для каждого элемента управления, достаточно просто щелкать мышью на элементах управления в выбранном нами порядке перехода. Для элементов управления, помещенных в группирующую рамку, *Tab Order Wizard* создает отдельную последовательность перехода.

Элемент управления MonthCalendar

В пространстве имен *System.Windows.Forms* предусмотрен элемент управления, при помощи которого пользователь может выбрать дату или диапазон дат, используя дружественный и удобный интерфейс. Это элемент управления MonthCalendar.

Наиболее важные свойства MonthCalendar представлены в табл. 7.11.

Таблица 7.11. Свойства MonthCalendar

Свойство	Назначение
BoldedDates	Массив объектов DateTime, выделенных подсветкой
CalendarDimensions	Определяет количество выводимых строк и столбцов

FirstDayOfWeek	Определяет, с какого дня будет начинаться неделя в MonthCalendar
MaxDate	Самая поздняя дата, которую разрешается выбрать пользователю (по умолчанию ограничений нет)
MaxSelectionCount	Максимальное количество дат, которое одновременно может выбрать пользователь
MinDate	Самая ранняя дата, которую разрешается выбрать пользователю (по умолчанию ограничений нет)
MonthlyBoldedDates	Массив выделенных подсветкой объектов DateTime для месяца
SelectionRange	Диапазон выделенных объектов
SelectionEnd	Самая поздняя дата в диапазоне выделенных объектов
SelectionStart	Самая ранняя дата в диапазоне выделенных объектов
ShowToday	Определяет, будет ли MonthCalendar выводить информацию о текущей дате
ShowTodayCircle	Определяет, будет ли MonthCalendar выводить информацию о текущей дате в нижней части и выделять ее в календаре обводкой
ShowWeekNumbers	Определяет, будет ли MonthCalendar отображать номера недель справа от каждой строки
TodayDate	Дата, которая будет считаться MonthCalendar сегодняшней. По умолчанию TodayDate - это системная дата на момент создания объекта MonthCalendar
TodayDateSet	Определяет, можно ли пользователю по своему усмотрению выбирать сегодняшнюю дату. Если для этого свойства установлено значение true, пользователь может выбрать в

	качестве сегодняшней (TodayDate) любое число

По умолчанию всегда выделяется (и подсветкой, и обводкой) текущая дата. Пользователь может выбрать другую дату - в этом и есть смысл графического интерфейса MonthCalendar.

Можно получить дату, выбранную пользователем в MonthCalendar, при помощи свойства SelectionStart. Это свойство возвращает ссылку на объект DateTime, которая хранится в специальной переменной (d) При помощи набора свойств типа DateTime можно извлечь всю необходимую информацию в нужном нам формате.

При помощи свойств Month, Day и Year можно извлечь из объектов DateTime нужную информацию и сформировали текстовые строки. Это вполне допустимый подход. Дело в том, что дату в необходимом текстовом формате проще получить из DateTime при помощи специальных "форматирующих" свойств самих объектов DateTime. Набор таких свойств (и некоторые методы) представлен в таблице 7.12.

Таблица 7.12. Члены класса DateTime

Член	**Назначение**
Date	Позволяет получить информацию о дате (дата всегда отсчитывается от полуночи)
Day, Month, Year	Позволяют получить соответственно день, месяц и число из текущего объекта DateTime
DayOfWeek	Возвращает день недели для объекта DateTime
DayOfYear	Возвращает номер дня в году для объекта DateTime
Hour, Minute, Second, *Millisecond*	Возвращают информацию о часе, минутах, секундах и миллисекундах для объекта DateTime
MaxValue, MinValue	Возвращают минимальное и максимальное значения для DateTime
Now, Today	Эти два статических свойства типа DateTime позволяют получить информацию о текущей дате и времени (Now) или только о текущей дате (Today)

Ticks	Позволяет получить счетчик "тиков" (с интервалом в 100 наносекунд) для объекта DateTime
ToLongDateString( ), ToLongTimeString( ), ToShortDateString( ), ToShortTimeString( )	Преобразуют текущее значение объекта DateTime в разные виды текстового представления

При помощи вышеперечисленных членов можно значительно упростить вывод текстовой информации о дате.

Элемент управления Panel

Назначение элемента управления Panel (панель) - с его помощью можно объединить прочие элементы управления на форме. Panel происходит от базового класса ScrollableControl и поддерживает полосы прокрутки.

Элементы управления Panel обычно используются для экономии пространства на форме. Например, если элементы управления, которые планируем разместить на форме, на ней не умещаются, то можно поместить их внутрь Panel и установить для свойства AutoScroll объекта Panel значение true. В результате пользователь получит возможность доступа к "не вмещающимся" элементам управления с помощью полос прокрутки.

Всплывающие подсказки (ToolTips)

Большинство приложений с современным пользовательским интерфейсом поддерживают всплывающие подсказки. В приложениях *.NET* эта возможность реализуется при помощи типа System.Windows.Forms.ToolTip. ToolTip (всплывающие подсказки) - это небольшие окна с текстом, появляющиеся при наведении указателя мыши на элемент управления на форме. Наиболее важные члены класса ToolTip представлены в таблице 7.13.

Таблица 7.13. Члены класса ToolTip

Член	**Назначение**
Active	Определяет, будет ли всплывающая подсказка активной. Возможность отключить всплывающие подсказки может быть полезной, например, если в приложении предусмотрено два варианта интерфейса: для обычных и для опытных пользователей
AutomaticDelay	Позволяет получить или установить время задержки (в миллисекундах) при появлении подсказки
AutoPopDelay	Время (в миллисекундах), в течение которого подсказка остается видимой, если указатель мыши неподвижен и находится в области, занимаемой соответствующим элементом управления. По умолчанию

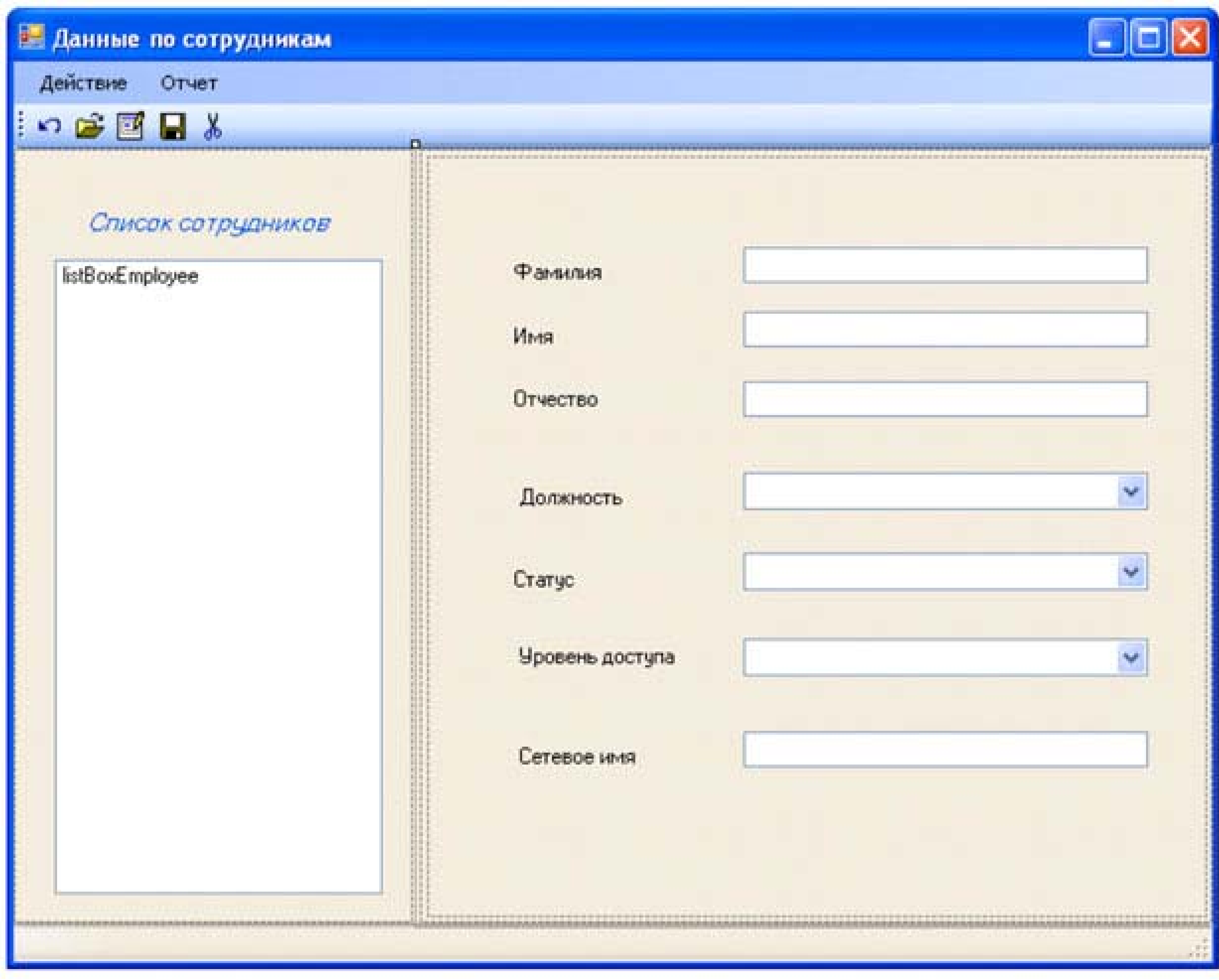
	это значение равно 10 значениям `AutomaticDelay`
`GetTooltip( )`	Возвращает текст подсказки
`InitialDelay`	Время (в миллисекундах), в течение которого указатель должен оставаться неподвижным в соответствующей области для появления подсказки. Значение по умолчанию равно значению AutomaticDelay
`ReshowDelay`	Время (в миллисекундах), в течение которого появится другая подсказка при перемещении указателя мыши от одного элемента управления к другому. По умолчанию это значение равно 1/5 от значения `AutomaicDelay`
`SetToolTip( )`	Ассоциирует подсказку с элементом управления

Для того чтобы настроить использование всплывающих подсказок для элементов управления можно сделать это с помощью графических средств *Visual Studio*.

Первое, что необходимо сделать - добавить на форму объект *ToolTip*, выбрав его в *ToolBox*. Затем можно указать текст всплывающей подсказки для любого элемента управления на форме (в том числе и для самой формы) из окна свойств данного элемента.

Проектирование элементов управления формы FormEmployee

Для разработки проекта приложения форма *FormEmployee* должна содержать *элементы управления*, в соответствии с видом, приведенном на рисунке 7.4 .



увеличить изображение
Рис. 7.4. Вид экранной формы FormEmployee

На данной форме необходимо сформировать *элементы управления*, приведенные в таблице 7.14.

Таблица 7.14. Элементы управления формы FormEmployee

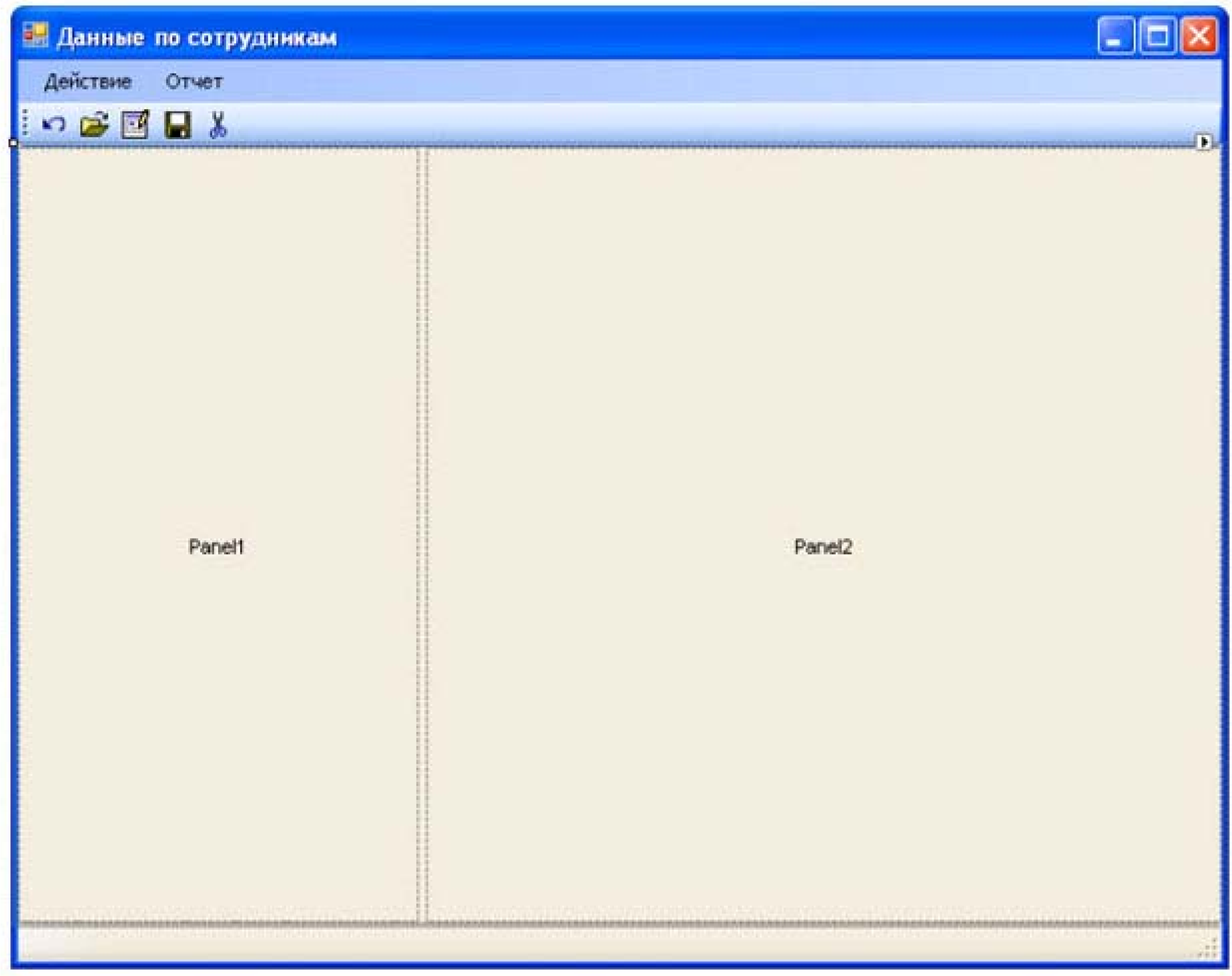
Элемент контроля	Имя	Свойство Text/Items	Назначение
SplitContainer	splitContainerEmployee		Две панели с разделителем
Label	labelListEmployee	Список сотрудников	Надпись
listBox	listBoxEmployee		Список сотрудников

Label	labelSurname	Фамилия	Надпись
Label	labelName	Имя	Надпись
Label	labelPatronymic	Отчество	Надпись
Label	labelJobRole	Должность	Надпись
Label	labelStatus	Статус	Надпись
Label	labelAccess	Уровень доступа	Надпись
label	labelNetName	Сетевое имя	Надпись
textBox	textBoxNetName		Сетевое имя
textBox	textBoxSurname		Фамилия
textBox	textBoxName		Имя
textBox	textBoxPatronymic		Отчество
comboBox	comboBoxJobRole		Должность
comboBox	comboBoxStatus	Активен, выходной, в отпуске, болеет, не работает, помечен как удаленный	Статус
comboBox	comboBoxAccess	Оператор, старший оператор, начальник смены, администратор, аналитик	Уровень доступа
menuItem	menuItemAction	Действие	Пункт меню

			"Редактировать"
menuItem	menuItemUndo	Отменить	Подпункт меню "Отменить"
menuItem	menuItemNew	Создать	Подпункт меню "Новый"
menuItem	menuItemEdit	Изменить	Подпункт меню "Изменить"
menuItem	menuItemSave	Сохранить	Подпункт меню "Сохранить"
menuItem	menuItem	Удалить	Подпункт меню "Удалить"
menuItem	menuItemReport	Отчет	Пункт меню "Отчет"
menuItem	menuItemReport1	По сотруднику	Подпункт меню "Отчет по сотруднику"
menuItem	menuItemReport2	По всем сотрудникам	Подпункт меню "Отчет по всем сотрудникам"

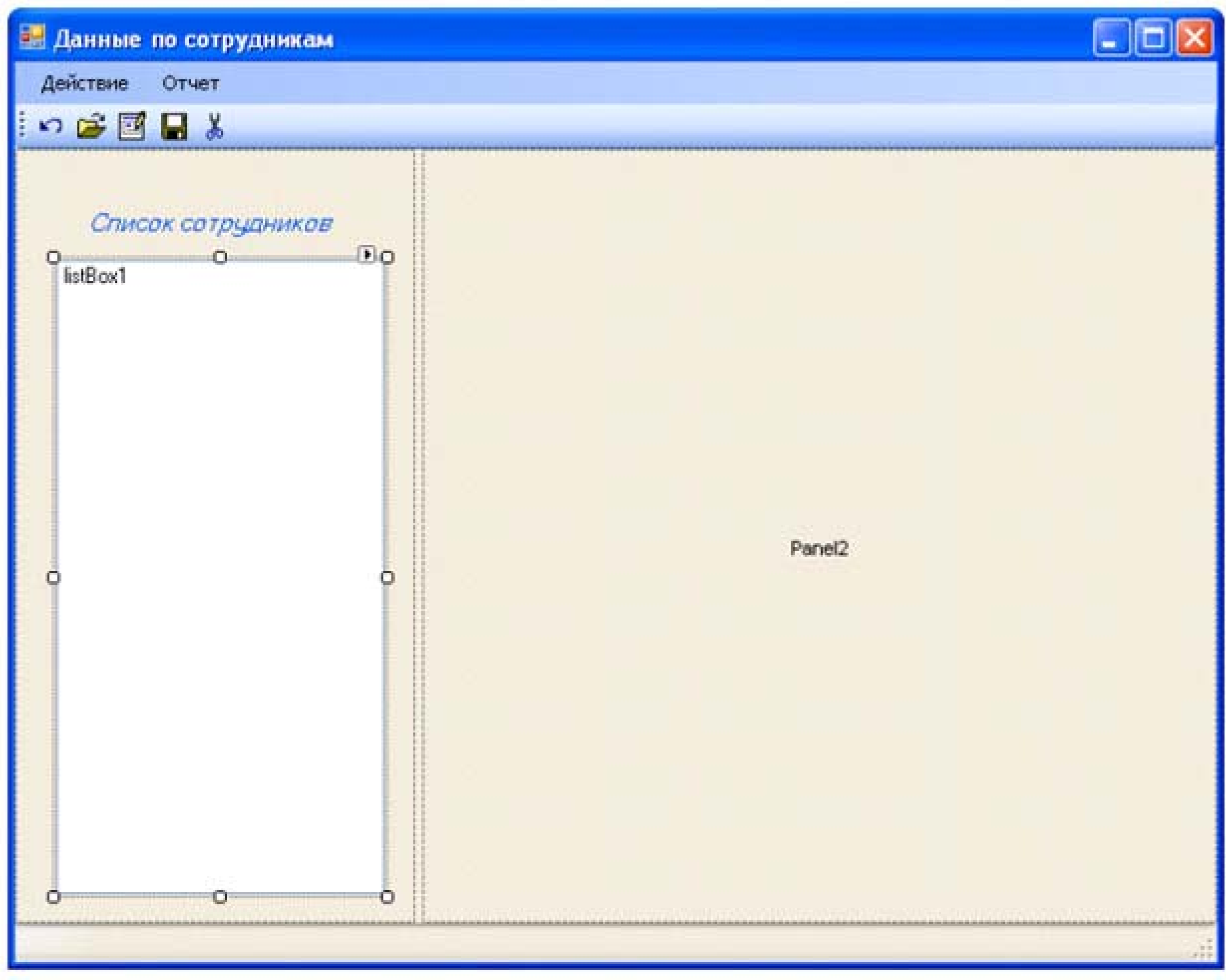
Вначале создайте на форме элемент *SplitContainer* (рисунок 7.5). На панели 1 создайте *элементы управления* labelListEmployee и listBoxEmployee (рисунок 7.6), а остальные *элементы управления*, приведенные в таблице 7.14, - на панели 2 (рисунок 7.4).

После создания на форме *FormEmployee* элементов управления в соответствии с таблицей 7.14 необходимо настроить порядок перехода между ними при нажатии клавиши *Tab*.



увеличить изображение

Рис. 7.5. Создание панелей на форме FormEmployee



увеличить изображение

Рис. 7.6. Формирование элементов управления на панели 1 формы FormEmployee

Для этого необходимо задать последовательные номера свойству `TabIndex` элементов управления (в разрабатываемой форме это необходимо сделать для элементов управления `TextBox` и `ComboBox`) из окна *Properties* (рисунок 7.7) или вызвать мастер `Tab Order Wizard` из *меню View/Tab Order* (рисунок 7.8). Задание последовательности значений свойству `TabIndex` производится щелчком мыши на элементах управления в заданной последовательности.

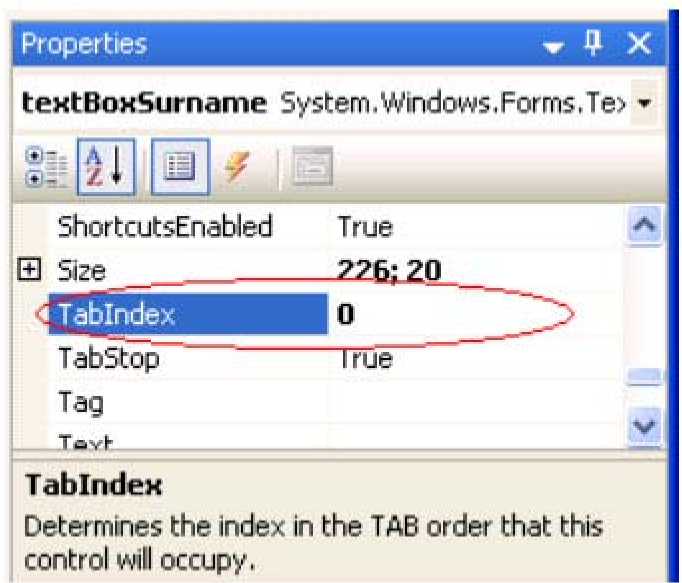
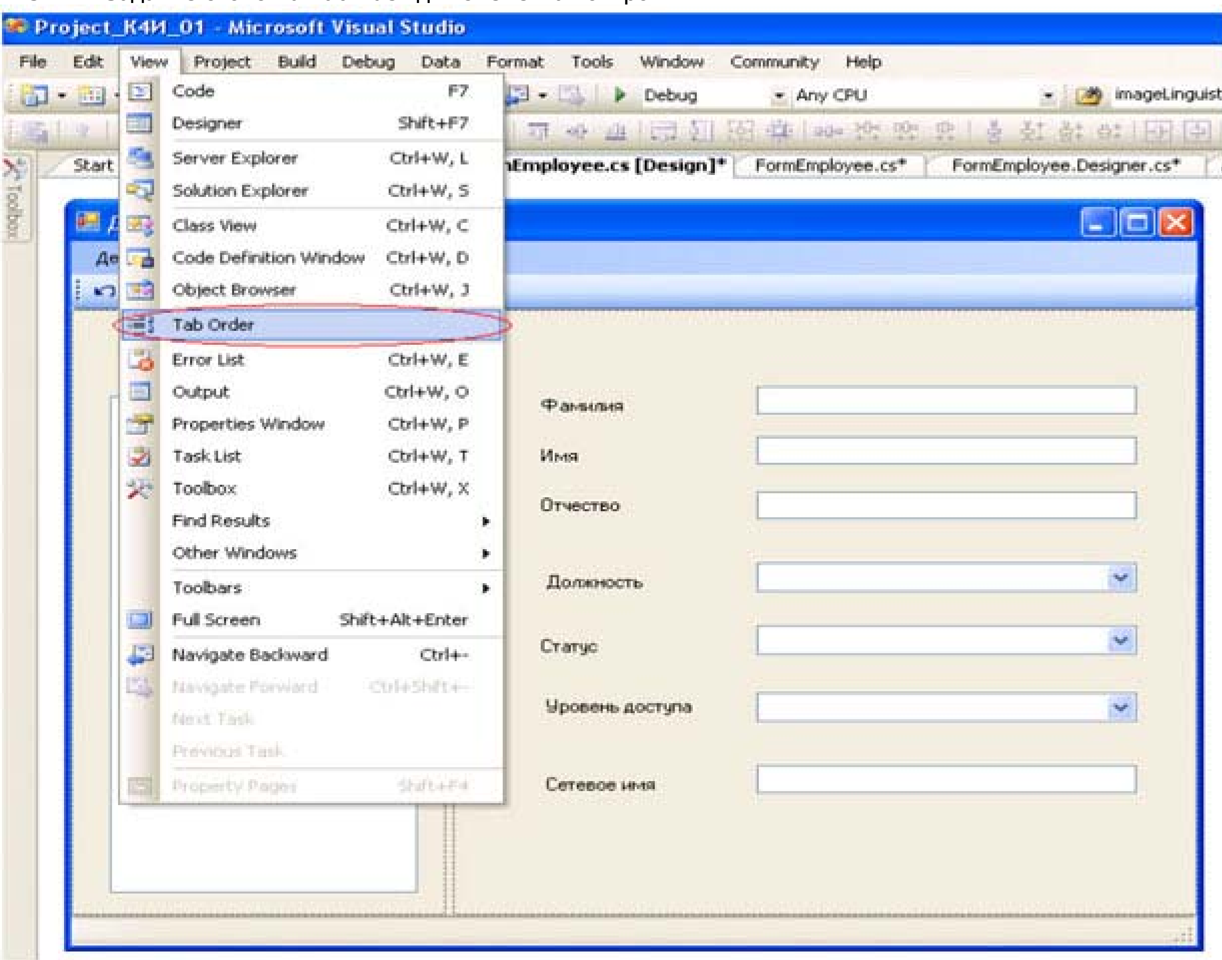


Рис. 7.7. Задание свойства TabIndex для элемента контроля

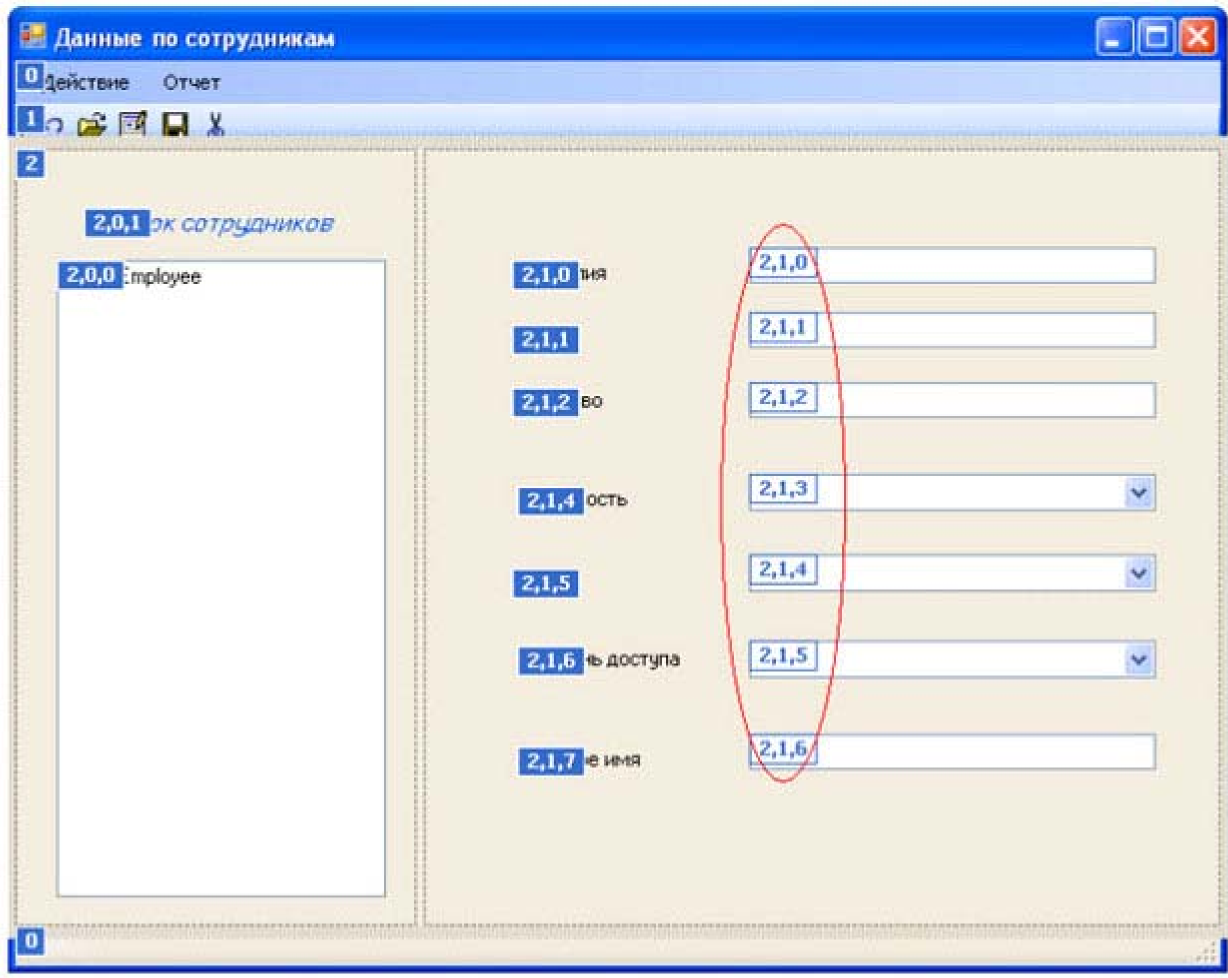


увеличить изображение

Рис. 7.8. Настройка перехода по элементам управления

Результат настройки порядка перехода между элементами управления при нажатии клавиши *Tab* приведен на рисунке 7.9.

Для работы с формой необходимо создать методы, которые разрешают только просматривать форму (режим просмотра) и редактировать форму (режим редактирования).



увеличить изображение

Рис. 7.9. Результат работы мастера Tab Order Wizard

Создадим метод для задания режима просмотра формы `DisplayReadOnly`. Метод `DisplayReadOnly` должен быть общедоступным, ничего не должен возвращать и не иметь параметров. Для задания режима просмотра (только для чтения) объекту класса `TextBox` необходимо свойству `ReadOnly` присвоить*значение* `true`, а для объекта класса `comboBox` - свойству `Enabled` *значение* `false`. Код метода `DisplayReadOnly` представлен далее:

```
public void DisplayReadOnly()
{
    this.textBoxSurname.ReadOnly = true;
    this.textBoxName.ReadOnly = true;
    this.textBoxPatronymic.ReadOnly = true;
    this.textBoxNetName.ReadOnly = true;
    this.comboBoxJobRole.Enabled = false;
    this.comboBoxStatus.Enabled = false;
    this.comboBoxAccess.Enabled = false;
}
```

Аналогичным образом сформируем метод *`DisplayEdit`*, который задает режим редактирования формы:

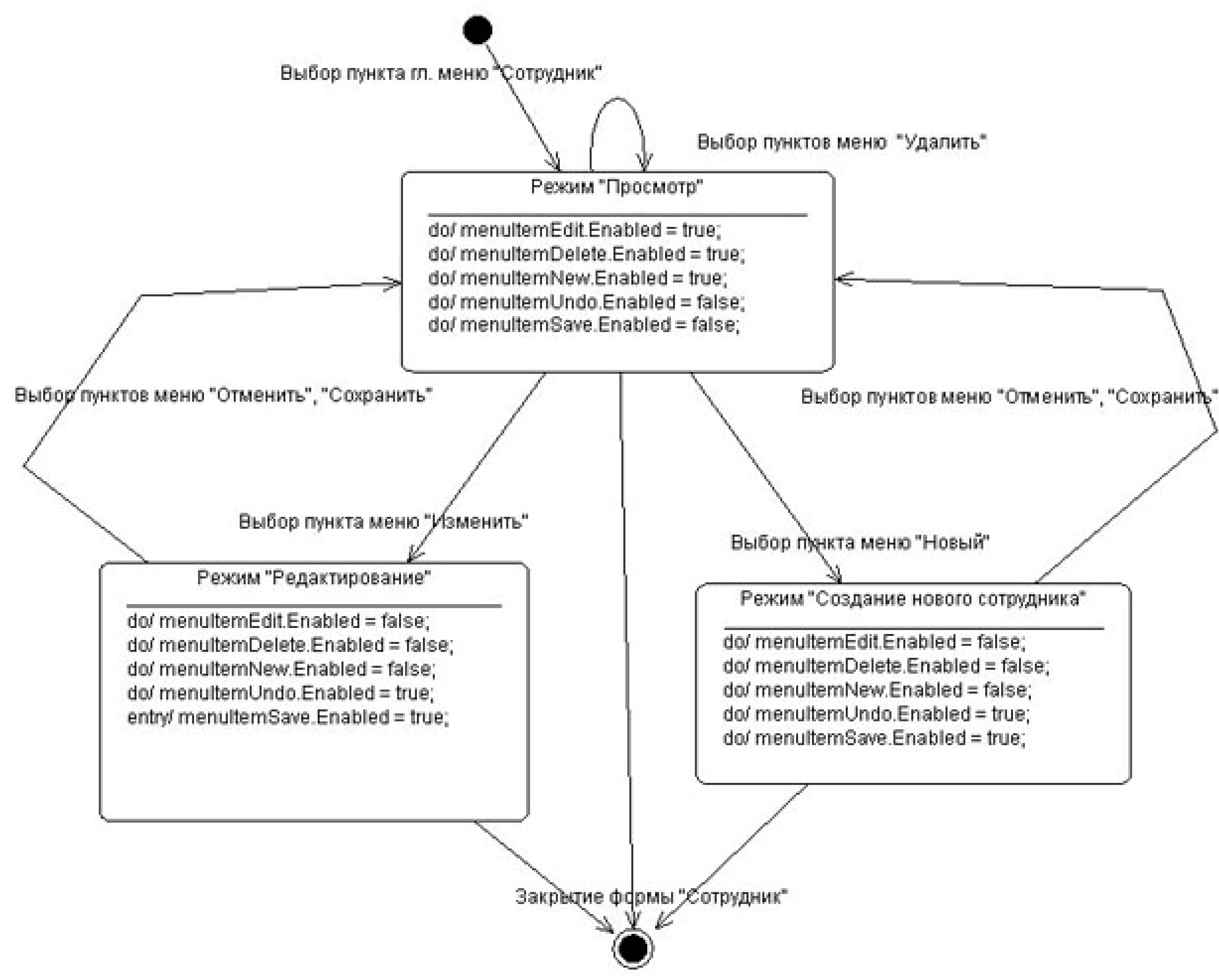
```
/// Задание режима редактирования
public void DisplayEdit()
{
    this.textBoxSurname.ReadOnly = false;
    this.textBoxName.ReadOnly = false;
    this.textBoxPatronymic.ReadOnly = false;
    this.textBoxNetName.ReadOnly = false;
    this.comboBoxJobRole.Enabled = true;
    this.comboBoxStatus.Enabled = true;
    this.comboBoxAccess.Enabled = true;
}
```

Для управления режимом доступности (только для чтения/редактирование) формы *FormEmployee* необходимо метод `DisplayReadOnly` вызывать при первоначальной загрузке формы (событие `Load`), при создании новых данных по сотруднику и при редактировании данных по сотруднику, а метод *`DisplayEdit`* - при сохранении данных по сотруднику и при отмене режима редактирования данных.

Проверьте правильность режима управления доступностью элементов управления формы *FormEmployee*.

Анализ кодов методов `DisplayReadOnly( )` и *`DisplayEdit( )`* показывает, что они могут быть объединены в один метод с параметром. Необходимо самостоятельно написать объединенный метод, получив в результате метод `DisplayReadOnly(bool readOnly)`, в котором параматр `readOnly` определяет режим редактирования: если `readOnly` равен `true`, то режим только для просмотра, если равен `false`, то - редактирование.

В процессе работы приложения необходимо управлять доступом к пунктам *меню* в соответствии с диаграммой состояний для пунктов *меню* формы *FormEmployee*, приведенной на рисунке 7.10.



увеличить изображение

Рис. 7.10. Диаграмма состояний для активности подпунктов меню "Действие"

Диаграмма отображает возможные переходы между тремя режимами: "Просмотр", "Редактирование" и "Создание нового сотрудника".

При выборе в главном *меню* приложения пункта "Сотрудник" *Windows* -форма *FormEmployee* должна перейти в режим "Просмотр", что определяет *доступ* к пунктам *меню* "Создать", "Редактировать", "Удалить" и запрет доступа к подпунктам *меню* "Отменить", "Сохранить".

Если в режиме просмотр выбирается подпункт *меню* "Удалить", то в результате выполнения данной функции режим *Windows*-формы *FormEmployee* не должен измениться, т.е. форма должна остаться в режиме "Просмотр".

Если в режиме просмотр выбирается подпункт *меню* "Изменить", то *Windows*-формы FormEmployee должна перейти в режим "Редактирование". Данный режим предполагает, что разрешается *доступ* к подпунктам *меню* "Отменить", "Сохранить" и запрещается доступа к подпунктам *меню* "Создать", "Редактировать", "Удалить".

Аналогичным образом интерпретируются переходы формы *FormEmployee* из одного режима в другой.

На рисунке 7.10 представлены режимы и переходы для подпунктов главного *меню*. Аналогичные режимы необходимо соблюдать для контекстного *меню* и кнопок панели инструментов.

Для управления доступом к пунктам главного *меню* создайте методы `MenuItemEnabled(bool itemEnabled)`, для контекстного *меню* - `MenuItemContextEnabled(bool itemEnabled)` и для кнопок

панели управления - `StripButtonEnabled(bool itemEnabled)`. Управление доступностью пунктов главного и контекстного *меню* осуществляется через свойство `Enabled` класса `ToolStripMenuItem`, а кнопок панели управления - через свойство `Enabled` класса `ToolStripButton`.

Проверьте правильность режима управления пунктов главного и контекстного *меню*, а также кнопок панели управления формы *FormEmployee*.

С учетом того, что установка режимов просмотра и редактирования экранной формы, а также *управление доступом* к пунктам *меню* должно выполняться при реализации нескольких функций программы целесообразно для избежания дублирования кода все методы управления режимами объединить в один метод`DisplayForm`.

```
private void DisplayForm(bool mode)
{
    DisplayReadOnly(mode);
    MenuItemEnabled(mode);
    MenuItemContextEnabled(mode);
    StripButtonEnabled(mode);
}
```

Первоначальная установка режима *"Просмотр"* должна проводиться при первоначальной загрузке формы *FormEmployee*.

Задание на практическое занятие

1. Изучить теоретический материал.
2. Для формы *FormEmployee* создать требуемые элементы контроля.
3. Разработать методы для задания режимов "Просмотр", "Редактирование" для элементов контроля.
4. Разработать методы для задания режимов "Просмотр", "Редактирование" для управления активностью пунктов главного меню формы, контекстного меню и кнопок панели инструментов.
5. Сформировать обработчик события `Load`.
6. Протестировать программу.

Практическое занятие 8. Подготовка ADO.NET к работе в приложении

Цель занятия: Изучить назначение и основные способы создания объектов *ADO.NET* при помощи *Visual Studio IDE*

Общие сведения

В платформе *.NET* определено множество типов (организованных в соответствующие пространства имен) для взаимодействия с локальными и удаленными хранилищами данных. Общее название пространств имен с этими типами - *ADO.NET*.

ADO.NET - это новая технология доступа к базам данных, специально оптимизированная для нужд построения рассоединенных (*disconnected*) систем на платформе *.NET*.

Технология *ADO.NET* ориентирована на приложения *N-tier* - архитектуру многоуровневых приложений, которая в настоящее время стала фактическим стандартом для создания распределенных систем.

Основные отличительные особенности *ADO.NET*:

- *ADO* расширяет концепцию объектов-наборов записей в базе данных новым типом *DataSet*, который представляет локальную копию сразу множества взаимосвязанных таблиц. При помощи объекта *DataSet* пользователь может локально производить различные операции с содержимым базы данных, будучи физически рассоединен с СУБД, и после завершения этих операций передавать внесенные изменения в базу данных при помощи соответствующего "адаптера данных" (`data adapter`);
- в *ADO.NET* реализована полная поддержка представления данных в *XML* -совместимых форматах. В *ADO.NET* сформированные для локальной обработки наборы данных представлены в формате *XML* (в этом же формате они и передаются с сервера баз данных). Данные в форматах *XML* очень удобно передавать при помощи обычного *HTTP*, решает многие проблемы с установлением соединений через брандмауэры;
- *ADO.NET* - это библиотека управляемого кода и взаимодействие с ней производится как с обычной сборкой *.NET*. Типы *ADO.NET* используют возможности управления памятью *CLR* и могут использоваться во многих *.NET* - совместимых языках. При этом обращение к типам *ADO.NET* (и их членам) производится практически одинаково вне зависимости от того, какой язык используется.

Все типы *ADO.NET* предназначены для выполнения единого набора задач:

- установить соединение с хранилищем данных;
- создать и заполнить данными объект *DataSet* ;
- отключиться от хранилища данных и вернуть изменения, внесенные в объект *DataSet* обратно в хранилище данных.

Объект DataSet - это *тип данных*, представляющий локальный набор таблиц и информацию об отношениях между ними.

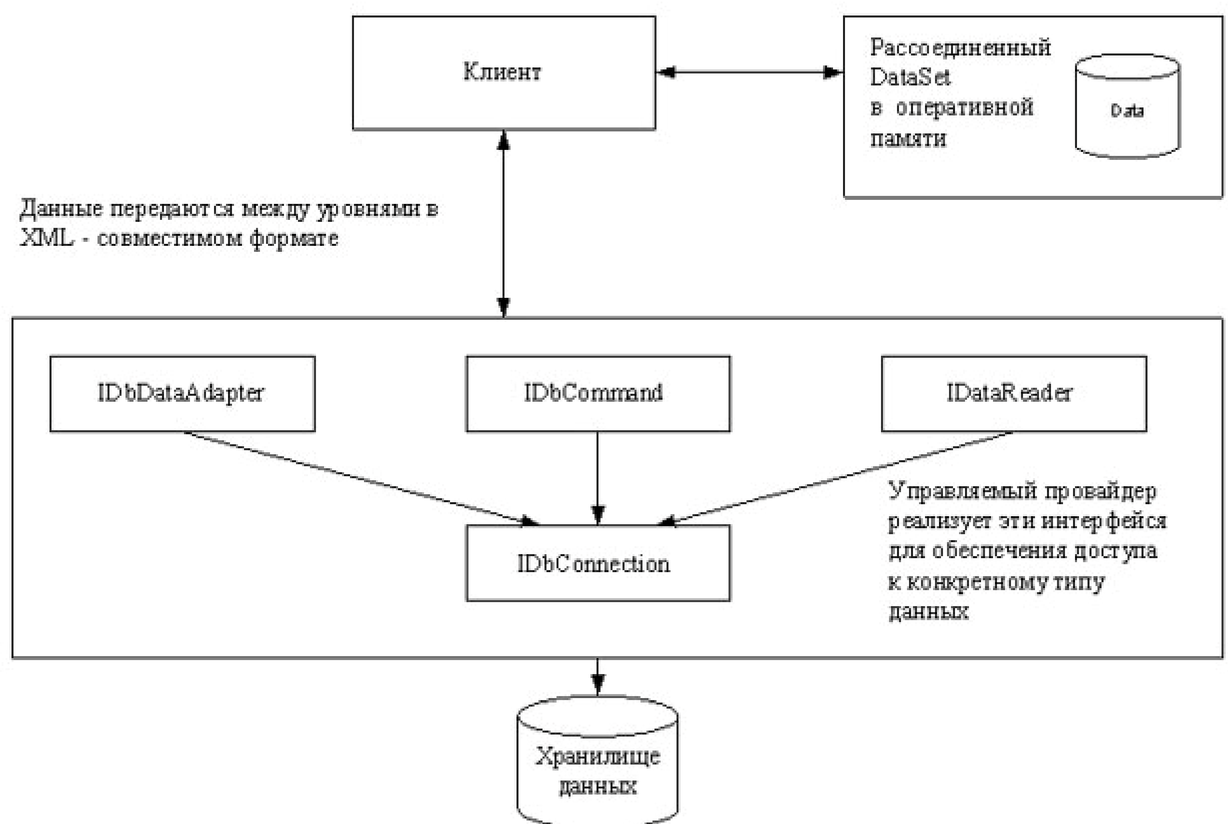
DataSet - набор связанных таблиц. На практике можно создать на клиенте *объект DataSet*, который будет представлять полную копию удаленной *базы данных*.

После создания объекта *DataSet* и его заполнения данными можно программными средствами производить запросы к нему и перемещаться *по* таблицам, выполнять все *операции*, как при работе с обычными базами данных: добавлять в таблицы новые записи, удалять и изменять существующие, применять к ним фильтры и т.п. После того как клиент завершит внесение изменений, *информация* о них будет отправлена в *хранилище данных* для обработки.

Создание *DataSet* осуществляется при помощи управляемого провайдера (`managed provider`).

Управляемый провайдер - это набор классов, реализующих интерфейсы, определенные в пространстве имен *System.Data*.

Речь идет об интерфейсах `IDbCommand`, `IDbDataAdapter`, `IDbConnection` и `IDataReader` (рисунок 8.1).



увеличить изображение

Рис. 8.1. Взаимодействие клиента с управляемыми провайдерами

В состав *ADO.NET* включены два управляемых провайдера: *провайдер SQL* и *провайдер OleDb*. *Провайдер SQL* специально оптимизирован под взаимодействие с *Microsoft SQL Server* версии 7.0 и последующих. Для других источников данных предлагается использовать *провайдер OleDb*, который можно использовать для обращения к любым хранилищам данных, поддерживающим протокол *OLE DB*. Следует отметить, что *провайдер OleDb* работает при помощи "родного" *OLE DB* и требует возможности взаимодействия при помощи *COM*.

Все возможности *ADO.NET* заключены в типах, определенных в соответствующих пространствах имен. Краткий обзор главных пространств имен *ADO.NET* представлен в таблице 8.1.

Таблица 8.1. Пространства имен ADO.NET

Пространство имен	Описание
System.Data	Главное пространство имен ADO.NET. В нем определены типы, представляющие таблицы, столбцы, записи, ограничения и тип - DataSet.

System.Data.Common	Определены типы, общие для всех управляемых провайдеров. Многие из них выступают в качестве базовых классов для классов из пространств имен для провайдеров SQL и OleDb
System.Data.OleDb	В этом пространстве имен определены типы для установления соединений с OLE DB-совместимыми источниками данных, выполнения к ним SQL-запросов и заполнения данными объектов DataSet.
System.Data.SqlClient	В этом пространстве имен определены типы, которые составляют управляемый провайдер SQL.
System.Data.SqlTypes	Представляют собой "родные" типы данных Microsoft SQL Server.

Все пространства имен *ADO.NET* расположены в одной сборке - `System.Data.dll`. Это означает, что в любом проекте, использующем *ADO.NET*, мы должны добавить ссылку на эту сборку.

В любом приложении *ADO.NET* необходимо использовать, *по* крайней мере, одно *пространство* имен - *System.Data*. Кроме того, практически во всех ситуациях требуется использовать либо *пространство* имен *System.Data.OleDb* или *System.Data.SqlClient* - для установления соединения с источником данных.

Типы пространства имен *System.Data* предназначены для представления данных, полученных из источника (но не для установления соединения непосредственно с источником).

В основном эти типы представляют собой *объектные представления* примитивов для работы с базами данных - таблицами, строками, столбцами, ограничениями и т. п. Наиболее часто используемые типы *System.Data* представлены в таблице 8.2.

Таблица 8.2. Типы пространства имен System.Data

Тип	Назначение
`DataColumnCollection`, `DataColumn`	`DataColumn` представляет один столбец в объекте `DataTable`, `DataColumnCollection` - все столбцы
`ConstraintCollection`, `Constraint`	`Constraint` - объектно-ориентированная оболочка вокруг ограничения (например, внешнего ключа или уникальности), наложенного на один или несколько `DataColumn`, `ConstraintCollection`- все ограничения в объекте `DataTable`

`DataRowCollection, DataRow`	`DataRow` представляет единственную строку в `DataTable`, `DataRowCollection` - все строки в `DataTable`
`DataRowView, Data View`	`DataRowView` позволяет создавать настроенное представление единственной строки, `DataView` - созданное программным образом представление объекта `DataTable`, которое может быть использовано для сортировки, фильтрации, поиска, редактирования и перемещения
`DataSet`	Объект, создаваемый в оперативной памяти на клиентском компьютере. `DataSet` состоит из множества объектов `DataTable` и информации об отношениях между ними
`ForeignKeyConstraint, UniqueConstraint`	`ForeignKeyConstraint` представляет ограничение, налагаемое на набор столбцов в таблицах, связанных отношениями первичный - внешний ключ. `UniqueConstraint` - ограничение, при помощи которого гарантируется, что в столбце не будет повторяющихся записей
`DataRelationCollection, DataRelation, DataTableCollection, DataTable`	Тип `DataRelationCollection` представляет набор всех отношений (то есть объектов `DataRelation`) между таблицами в `DataSet`. Тип `DataTableCollection` представляет набор всех таблиц (объектов `DataTable`) в `DataSet`

В традиционных системах клиент-*сервер* при запуске приложения пользователем автоматически устанавливается *связь* с базой данных, которая поддерживается в "активном" состоянии до тех пор, пока *приложение* не будет закрыто. Такой метод работы с данными становится непрактичным, поскольку подобные приложения трудно масштабируются. Например, такая прикладная система может работать достаточно быстро и эффективно при наличии 8-10 пользователей, но она может стать полностью неработоспособной, если с ней начнут работать 100, 200 и более пользователей. Каждое открываемое соединение с базой данных "потребляет" достаточно много системных ресурсов сервера, они становятся занятыми поддержкой и обслуживанием открытых соединений, их не остается на процессы непосредственной обработки данных.

При разработке прикладных систем в сети *Интернет* (*Web* -приложения) необходимо добиваться максимальной масштабируемости. Система должна работать одинаково эффективно как с малым, так и с большим числом пользователей.

По этой причине, в *ADO.NET* используется модель работы пользователя в отрыве от источника данных. Приложения подключаются к базе данных только на небольшой промежуток времени. Соединение устанавливается только тогда, когда клиент удаленного компьютера запрашивает на сервере данные. После того, как*сервер* подготовил необходимый набор данных, сформировал и отправил их клиенту в виде *WEB* -

страницы, *связь* приложения с сервером сразу же обрывается, и клиент просматривает полученную информацию уже не в связи с сервером. При работе в сети *Интернет* нет необходимости поддерживать постоянную "жизнеспособность" открытых соединений, поскольку неизвестно, будет ли конкретный клиент вообще далее взаимодействовать с источником данных. В таком случае целесообразнее сразу освобождать занимаемые серверные ресурсы, что обеспечит обслуживание большего количества пользователей. Модели доступа к данным представлена на рисунке 8.2.

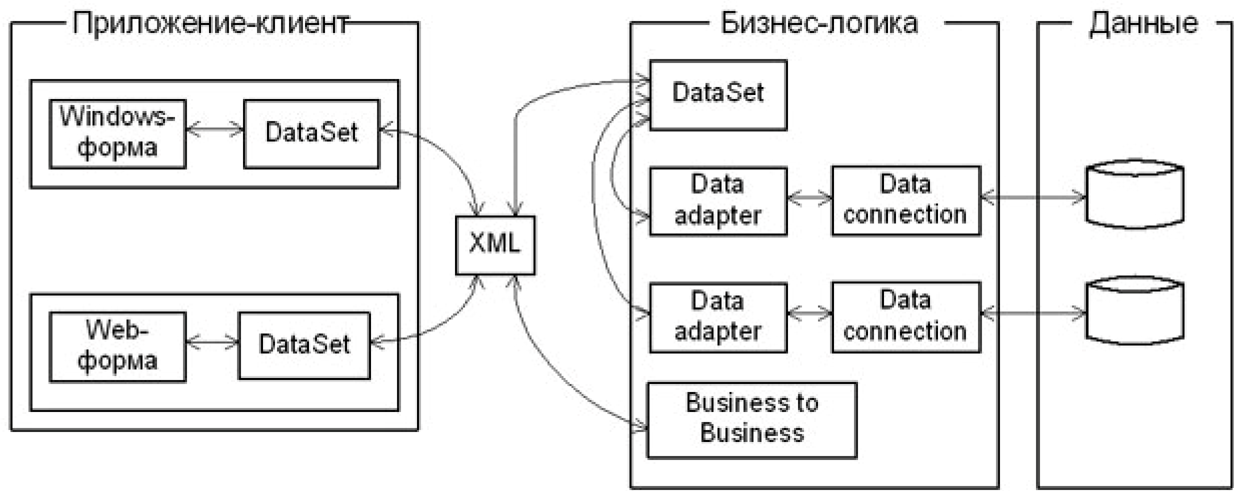


Рис. 8.2. Модель доступа к данным в ADO.NET

В объектной модели *ADO.NET* можно выделить несколько уровней.

Уровень данных. Это *по* сути дела базовый уровень, на котором располагаются сами данные (например, таблицы *базы данных MS SQL Server*). На данном уровне обеспечивается физическое хранение информации на магнитных носителях и манипуляция с данными на уровне исходных таблиц (*выборка*, *сортировка*, добавление, удаление, обновление и т. п.).

Уровень бизнес-логики. Это набор объектов, определяющих, с какой базой данных предстоит установить *связь* и какие действия необходимо будет выполнить с содержащейся в ней информацией. Для установления связи с базами данных используется *объект* `DataConnection`. Для хранения команд, выполняющих какие либо действия над данными, используется *объект* `DataAdapter`. И, наконец, если выполнялся процесс выборки информации из *базы данных*, для хранения результатов выборки используется *объект* `DataSet`. *Объект* `DataSet` представляет собой набор данных "вырезанных" из таблиц основного хранилища, который может быть передан любой программе-клиенту, способной либо отобразить эту информацию конечному пользователю, либо выполнить какие-либо манипуляции с полученными данными.

Уровень приложения. Это набор объектов, позволяющих хранить и отображать данные на компьютере конечного пользователя. Для хранения информации используется уже знакомый нам *объект* `DataSet`, а для отображения данных имеется довольно большой набор элементов управления (`DataGrid`, `TextBox`, `ComboBox`, `Label` и т. д.). В *Visual Studio .Net* можно вести разработку двух типов приложений. В первую *очередь* это традиционные *Windows* -приложения (на основе *Windows* -форм), которые реализованы в виде exe-файлов, запускаемых на компьютере пользователя. Ну и конечно, *Web* -приложения (на основе *Web* -форм), которые работают в оболочке браузера. Как видно из рисунка 8.2, для хранения данных на уровне обоих типов приложений используется *объект* `DataSet`.

Обмен данными между приложениями и уровнем бизнес-логики происходит с использованием формата *XML*, а средой передачи данных служат либо локальная *сеть* (*Интранет*), либо глобальная *сеть* (*Интернет*).

В *ADO.NET* для манипуляции с данными могут использоваться команды, реализованные в виде *SQL* -запросов или хранимых процедур (`DataCommand`). Например, если необходимо получить некий набор информации *базы данных*, вы формируете команду `SELECT` или вызываете хранимую процедуру *по* ее имени.

Когда требуется получить набор строк из *базы данных*, необходимо выполнить следующую последовательность действий:

- открыть соединение (`connection`) с базой данных;
- вызвать на исполнение метод или команду, указав ей в качестве параметра текст *SQL* -запроса или имя хранимой процедуры;
- закрыть соединение с базой данных.

Связь с базой данных остается активной только на достаточно короткий срок - на период выполнения запроса или хранимой процедуры.

Когда *команда* вызывается на *исполнение*, она возвращает либо данные, либо *код ошибки*. Если в команде содержался *SQL -запрос* на выборку - `SELECT`, то *команда* может вернуть набор данных. Вы можете выбрать из *базы данных* только определенные строки и колонки, используя *объект* `DataReader`, который работает достаточно быстро, поскольку использует курсоры `read-only, forward-only`.

Если требуется выполнить более чем одну операцию с данными, например, получить некоторый набор данных, а затем скорректировать его, - то необходимо выполнить последовательность команд. Каждая *команда* выполняется отдельно, последовательно одна за другой. Между выполняемыми командами соединение с базой отсутствует. Например, чтобы получить данные из базы - открывается *связь*, выбираются данные, затем *связь* закрывается. Когда выполняется обновление базы после корректировки информации пользователем, снова открывается *связь*, выполняется обновление данных в исходных таблицах и *связь* снова закрывается.

Команды работы с данными могут содержать параметры, т. е. могут использоваться параметризированные запросы, как, например, следующий *запрос*:

```
SELECT * FROM customers WHERE (customer_id=@customerid)
```

Значения параметров могут задаваться динамически, во *время выполнения* приложения.

Как правило, в приложениях необходимо извлечь информацию из *базы данных* и выполнить с ней некоторые действия: показать пользователю на экране монитора, сделать нужные расчеты или послать данные в другой *компонент*. Очень часто, в приложении нужно обработать не одну *запись*, а их набор: *список*клиентов, перечень заказов, набор элементов заказа и т. п. Как правило, в приложениях требуется одновременная работа с более чем одной таблицей: клиенты и все их заказы; *автор* и все его книги, заказ и его элементы, т.е. с набором связанных данных. Причем для удобства пользователя данные требуется группировать и сортировать то *по* одному, то *по* другому признаку. При этом нерационально каждый раз возвращаться к исходной базе данных и заново считывать данные. Более практично работать с некой временной "вырезкой" информации, хранящейся в оперативной памяти компьютера.

Эту роль выполняет набор данных - `DataSet`, который представляет собой своеобразный *кэш* записей, извлеченных из базового источника. `DataSet` может состоять из одной или более таблиц, он имеет дело с копиями таблиц из *базы данных* источника. Кроме того, в данном объекте могут содержаться связи между таблицами и некоторые ограничения на выбираемые данные. Структура объекта `DataSet` приведена на рисунке 8.3.

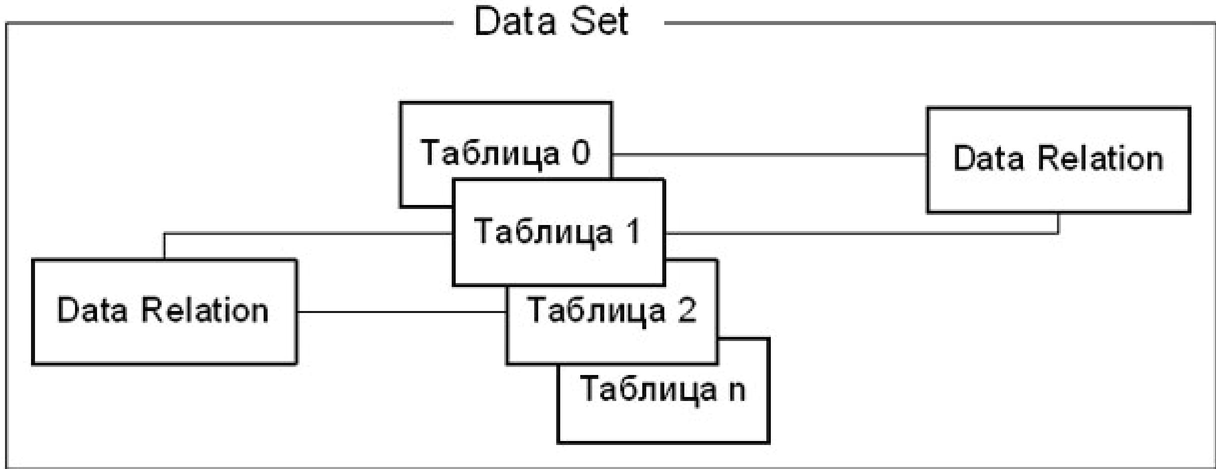


Рис. 8.3. Структура объекта DataSet

Данные в `DataSet` - это некий уменьшенный вариант основной *базы данных*. Тем не менее, вы можете работать с такой "вырезкой" точно так же, как и с реальной базой. Поскольку каждый *пользователь* манипулирует с полученной порцией информации, оставаясь отсоединенными от основной *базы данных*, последняя может в это время решать другие задачи.

Конечно, практически в любой задаче обработки данных требуется корректировать информацию в базе данных (хотя и не так часто, как извлекать данные из нее). Вы можете выполнить *операции* коррекции непосредственно в `DataSet`, а потом все внесенные изменения будут переданы в основную базу данных.

Важно отметить то, что `DataSet` - *пассивный контейнер* для данных, который обеспечивает только их хранение. Что же нужно поместить в этот *контейнер*, определяется в другом объекте - адаптере данных `DataAdapter`. В адаптере данных содержатся одна или более команд, которые определяют, какую информацию нужно поместить в таблицы объекта `DataSet`, *по* каким правилам нужно синхронизировать информацию в конкретной таблице `DataSet` и соответствующей таблицей основной *базы данных* и т. п. *Адаптер* данных обычно содержит четыре команды `SELECT, INSERT, UPDATE, DELETE`, для выборки, добавления, корректировки и удаления записей.

Например, метод `Fill` объекта `DataAdapter`, заполняющего данными *контейнер* `DataSet`, может использовать в элементе `SelectCommand` следующий *запрос*:

```
SELECT au_id, au_lname, au_fname FROM authors
```

Набор данных `DataSet` - "независимая" копия фрагмента *базы данных*, расположенная на компьютере пользователя. Причем в этой копии могут быть не отражены те изменения, которые могли внести в основную базу данных другие пользователи. Если требуется увидеть самые последние изменения, сделанные другими пользователями, то необходимо "освежить" `DataSet`, повторно вызвав метод `Fill` адаптера данных.

Информация о базе данных

Разрабатываемое *приложение* предназначено для работы с базой данных сотрудников компании. На рисунке 8.4 представлена структура *базы данных*.

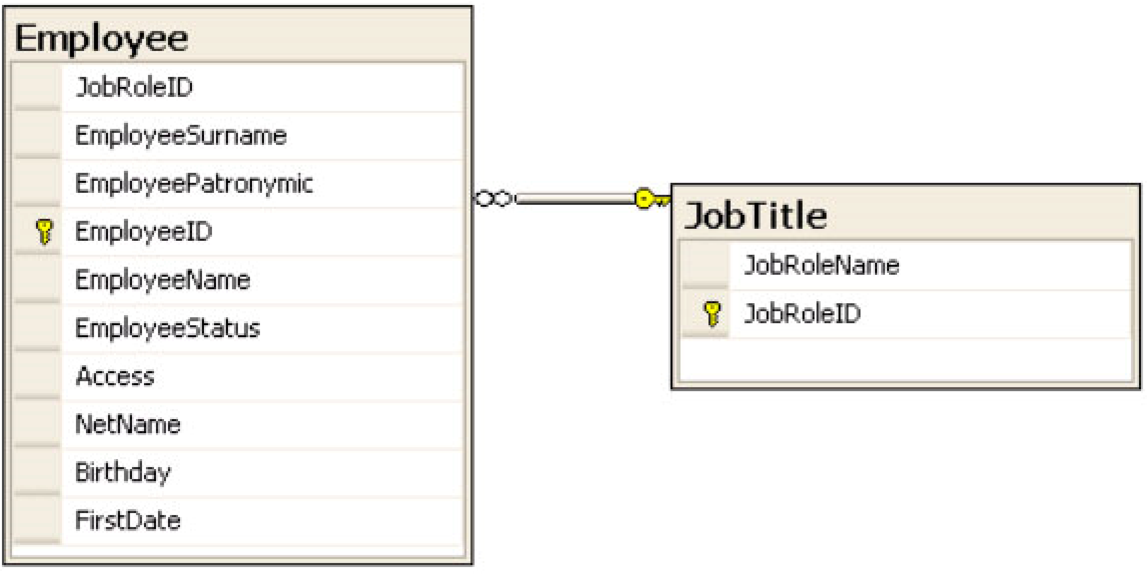


Рис. 8.4. Структура базы данных по сотрудникам компании

База данных включает две таблицы:

- сведения о сотрудниках - *Employee* ;
- справочник должностей - *JobTitle*.

Назначение атрибутов таблицы *Employee* приведены в таблице 8.4

Таблица 8.4. Атрибуты таблицы Employee

Имя атрибута	Назначение	Тип
EmployeeID	Суррогатный ключ	smallint
JobRoleID	Внешний ключ	smallint
EmployeeSurname	Фамилия	varchar(50)
EmployeeName	Имя	varchar(20)
EmployeePatronymic	Отчество	varchar(20)
EmployeeStatus	Статус	int

Access	Уровень доступа	varchar(20)
NetName	Сетевое имя	varchar(20)
Birthday	Дата рождения	Smalldatetime
FirstDate	Дата приема на работу	smalldatetime

Суррогатный *ключ* EmployeeID, как и все остальные суррогатные ключи *базы данных*, генерируется сервером *базы данных* автоматически, т.е. для него задано свойство IDENTITY для *СУБД MS SQL Server* или AutoNumber для *MS Access*. *Атрибут* JobRoleID является внешним ключом, с помощью которого осуществляется *связь* с таблицей *JobTitle*.

Назначение атрибутов таблицы *JobTitle* приведено в таблице 8.3.

Таблица 8.3. Атрибуты таблицы JobTitle

Имя атрибута	**Назначение**	**Тип**
JobRoleID	Суррогатный ключ	smallint
JobRoleName	Наименование должности	varchar(50)

В рассматриваемом приложении в качестве *СУБД* используется *MS SQL Server 2005*. Создаем соединение проекта с базой данных. Для этого выбираем *пункт меню Tools/Connect to Database*. Появляется окно *AddConnection* (рисунок 8.5)

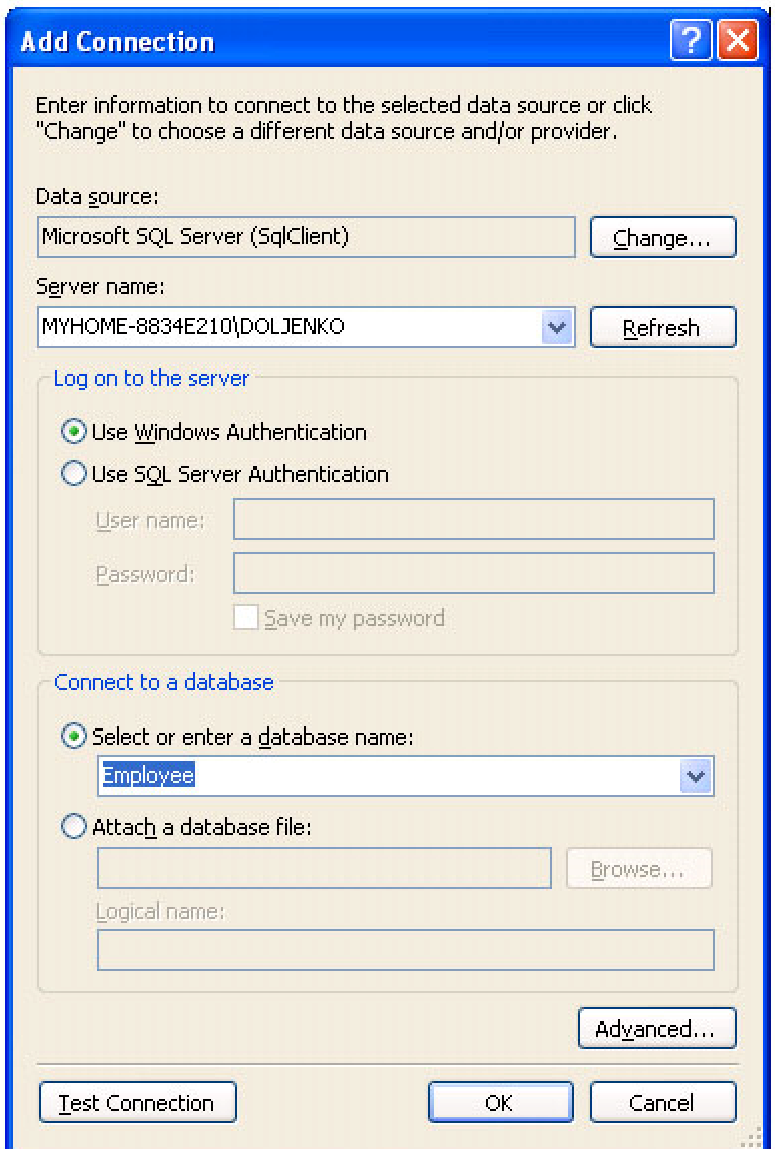


Рис. 8.5. Окно AddConnection

В пункте *"Server name"* задаем имя сервера, которое необходимо узнать у преподавателя (на рисунке 8.5 `MYHOME-8834E210\DOLJENKO`). В пункте *Select or enter database name* - имя *базы данных*, которое определит преподаватель (на рисунке 8.5 - `Employee`).

Для проверки правильности подключения к базе данных нажимаем клавишу *"Test Connection"*. При правильном подключении появляется следующее сообщение (рисунок 8.6).

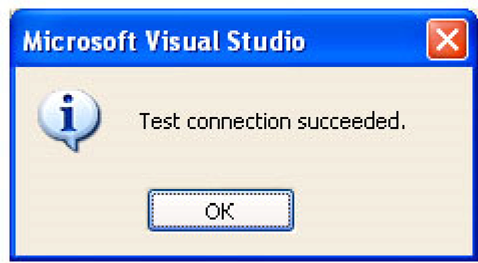


Рис. 8.6. Окно Microsoft Data Link

При нормальном соединении с базой данных можно открыть навигатор *Server Explorer* из *меню View/ Server Explorer* или сочетанием клавиш *ALT+CTRL+S* (рисунок 8.7).

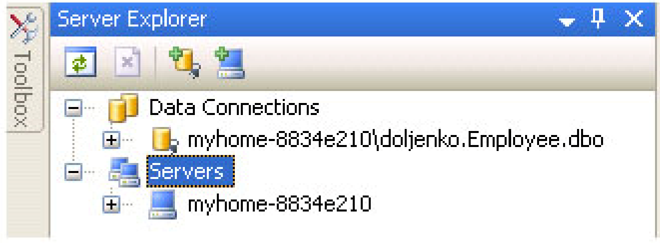


Рис. 8.7. Окно навигатора Server Explorer

Добавим в проект *объект* класса `DataSet`. Для этого выберем *пункт меню Project/Add New Item. . .* (рисунок 8.8).

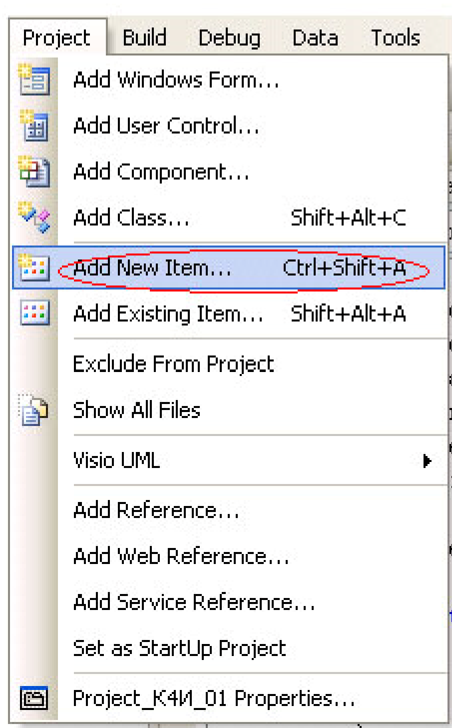
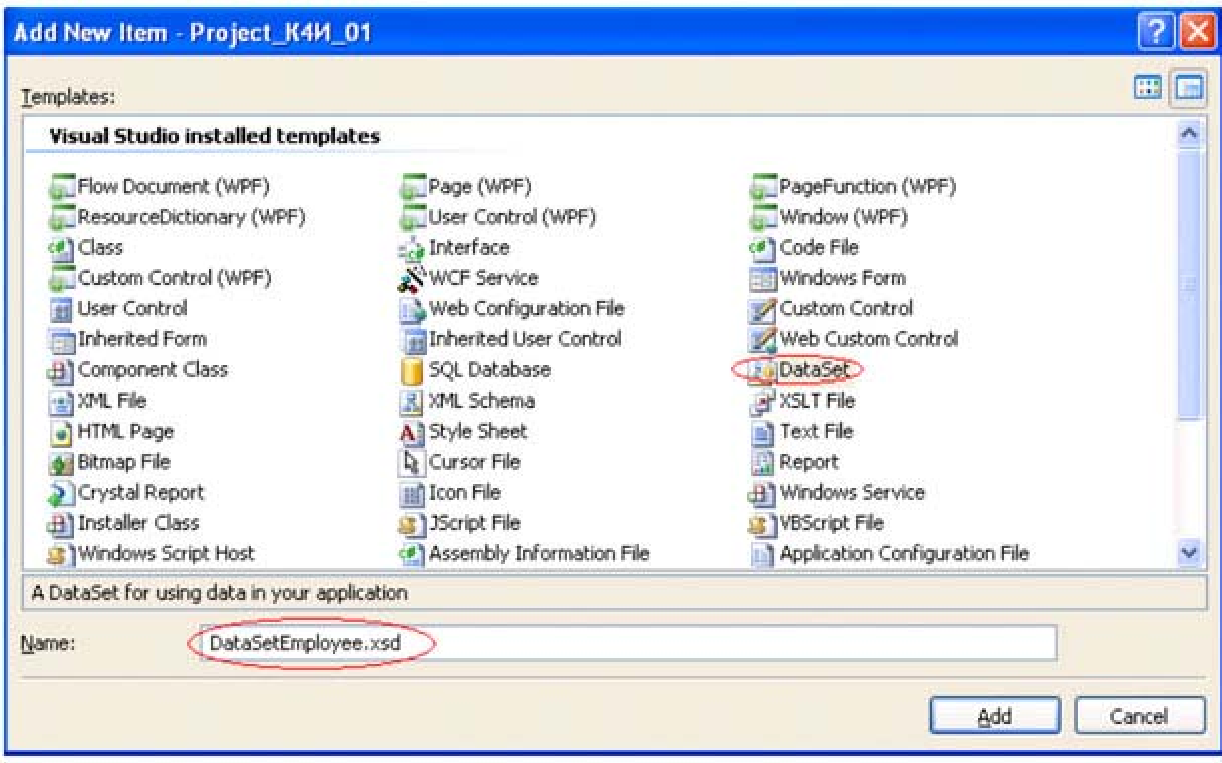


Рис. 8.8. Добавление в проект нового компонента

В окне *Add New Item* (рисунок 8.9) выберем *шаблон DataSet* и присвоим ему имя `DataSetEmployee`.



увеличить изображение

Рис. 8.9. Выбор нового компонента - DataSet

После нажатия кнопки *Add* система генерирует *класс* DataSetEmployee, который добавляется в решение проекта (рисунок 8.10).

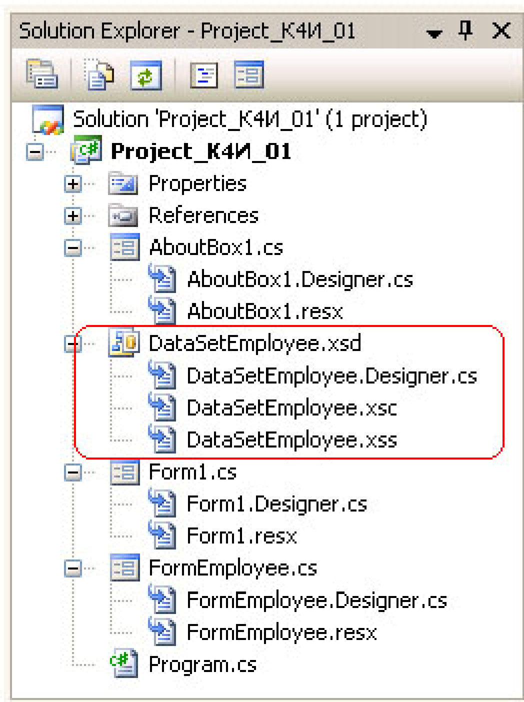
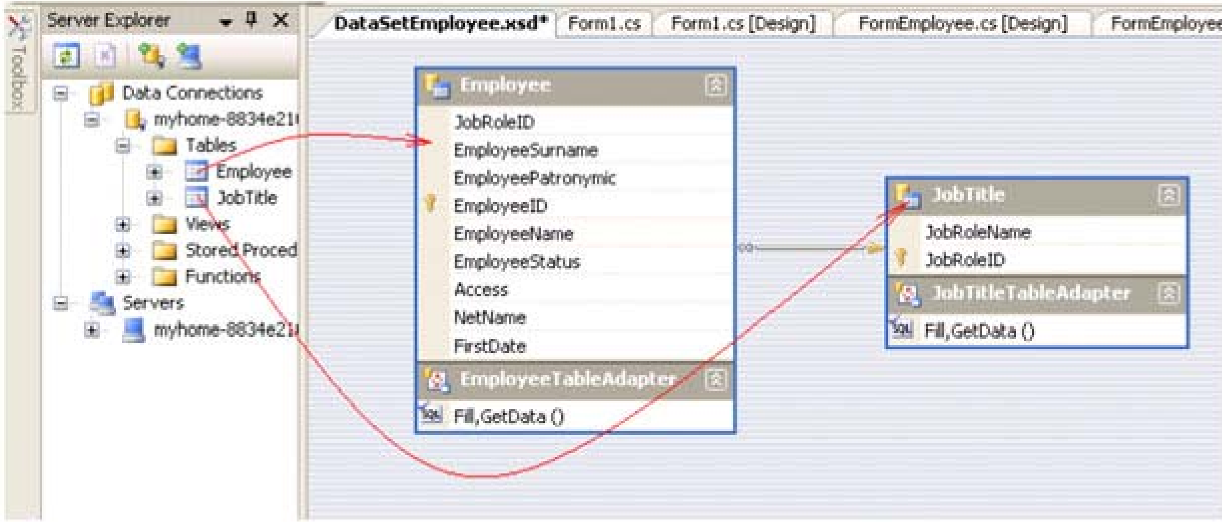


Рис. 8.10. Окно решения проекта с новым компонентом DataSet

Для добавления таблиц *Employee* и *JobTitle* к *DataSet* необходимо перетащить их из окна *Server Explorer* на *поле* графического дизайнера (рисунок 8.11).



увеличить изображение

Рис. 8.11. Добавление таблиц к DataSet

В результате будут созданы классы таблиц, адаптеры и методы `Fill` и `GetData`.

При формировании класса `DataSetEmployee` необходимо учесть то, что первичные ключи таблиц *Employee* и *JobTitle* являются суррогатными и автоматически формируются (*ключ* со свойством автоинкремент) источником данных (например, *MS SQL Server*). При формировании новых записей в приложении необходимо обеспечить уникальность первичных ключей для таблиц объекта `DataSetEmployee`. Это можно обеспечить, задав для ключевых колонок таблиц *Employee* и *JobTitle* следующие свойства:

```
AutoIncrement = true;
AutoIncrementSeed = -1;
AutoIncrementStep = -1;
```

Столбец со свойством `AutoIncrement` равным `true` генерирует последовательность значений, начинающуюся со значения `AutoIncrementSeed` и имеющую шаг `AutoIncrementStep`. Это позволяет генерировать уникальные значения целочисленного столбца первичного ключа. В этом случае при добавлении новой записи в таблицу будет генерироваться новое *значение* первичного ключа, начиная с -1, -2, -3 и т.д., которое никогда не совпадет с первичным ключом источника данных, т.к. в базе данных генерируются положительные первичные ключи.
Свойства `AutoIncrementSeed` и `AutoIncrementStep` устанавливаются равными -1, чтобы гарантировать, что когда набор данных будет синхронизироваться с источником данных, эти значения не будут конфликтовать со значениями первичного ключа в источнике данных. При синхронизации `DataSet` с источником данных, когда добавляют новую строку в таблицу *MS SQL Server 2005* с первичным автоинкрементным ключом, *значение*, которое этот *ключ* имел в таблице *DataSet*, заменяется значением, сгенерированным *СУБД*.

Установка свойств `AutoIncrement`, `AutoIncrementSeed` и `AutoIncrementStep` для колонки первичного ключа `EmployeeID` таблицы *Employee* приведена на рисунке 8.12.

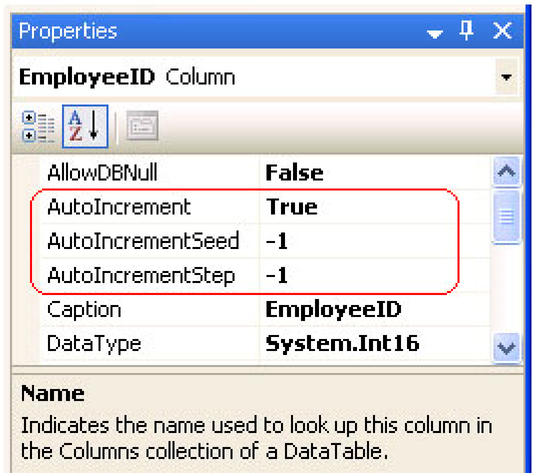


Рис. 8.12. Установка свойств для колонки EmployeeID

Аналогичные установки свойств AutoIncrement, AutoIncrementSeed и AutoIncrementStep необходимо сделать и для колонки JobTitleID таблицы *JobTitle*.

После создания класса DataSetEmployee и адаптера необходимо создать объекты этих классов, добавив следующий код к файлу FormEmployee.cs.

```
DataSetEmployee dsEmployee = new DataSetEmployee();
DataSetEmployeeTableAdapters.EmployeeTableAdapter daEmployee =
   new Project_K4И_01.DataSetEmployeeTableAdapters.
EmployeeTableAdapter();
DataSetEmployeeTableAdapters.JobTitleTableAdapter daJobTitle =
  new Project_K4И_01.DataSetEmployeeTableAdapters.
JobTitleTableAdapter();
```

После того, как созданы объекты адаптеров данных daEmployee и daJobTitle, а также *объект* класса DataSetEmployee - dsEmployee необходимо создать метод для заполнения объекта dsEmployee из *базы данных* (в рассматриваемом примере *база данных* Employee, созданная в *СУБД MS SQL Server 2005*). Для заполнения данными dsEmployee из *базы данных* Employee создадим метод EmployeeFill( ):

```
public void EmployeeFill()
{    daJobTitle.Fill(dsEmployee.JobTitle);
     daEmployee.Fill(dsEmployee.Employee);
     MessageBox.Show("Метод Fill отработал");}
```

В методе EmployeeFill( ) для объектов класса DataAdapter применяется метод Fill, который производит заполнение таблиц (*JobTitle* и *Employee*) объекта dsEmployee данными из *базы данных*. Метод Fill адаптера данных DataAdapter требует указания в качестве параметров задания соответствующей таблицы *DataSet*, то есть dsEmployee.JobTitle и dsEmployee.Employee.

Метод MessageBox.Show введен в метод EmployeeFill для первоначального тестирования, после которого его нужно убрать.

Вызов метода EmployeeFill необходимо добавить в обработчик события Load для формы FormEmployee, возникающего при нажатии на *пункт меню "Сотрудник"*.

Задание на практическое занятие

1. Изучите теоретический материал.
2. Создайте класс `DataSetEmployee`.
3. Для разрабатываемого приложения создайте объекты `dsEmployee`, `daJobTitle` и `daEmployee`.
4. Проведите компиляцию проекта и убедитесь в отсутствии ошибок трансляции.
5. Разработайте метод `Fill` для заполнение таблиц *DataSet*.
6. Протестировать работу приложения.

СПИСОК ЛИТЕРАТУРЫ

Перечень основной литературы:

1. Иванова Г.С. Объектно-ориентированное программирование [Электронный ресурс]: учебник/ Иванова Г.С., Ничушкина Т.Н. — Электрон. текстовые данные. — Москва: Московский государственный технический университет имени Н.Э. Баумана, 2014. — 456 с. — Режим доступа: http://www.iprbookshop.ru/94030.html. — ЭБС «IPRbooks».
2. Маляров А.Н. Объектно-ориентированное программирование [Электронный ресурс]: учебник для технических вузов/ Маляров А.Н.— Электрон. текстовые данные. — Самара: Самарский государственный технический университет, ЭБС АСВ, 2017.— 332 с.— Режим доступа: http://www.iprbookshop.ru/91772.html.— ЭБС «IPRbooks».
3. Мурадханов С.Э. Информатика и программирование: объектно-ориентированное программирование (на основе языка C#) [Электронный ресурс]: учебник/ Мурадханов С.Э., Широков А.И. — Электрон. текстовые данные. — Москва: Издательский Дом МИСиС, 2015. — 309 с. — Режим доступа: http://www.iprbookshop.ru/98855.html.— ЭБС «IPRbooks».

Перечень дополнительной литературы:

1. Зыков С.В. Введение в теорию программирования. Объектно-ориентированный подход [Электронный ресурс]: учебное пособие/ Зыков С.В.— Электрон. текстовые данные.— Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021.— 187 c.— Режим доступа: http://www.iprbookshop.ru/102007.html.— ЭБС «IPRbooks».
2. Николаев Е.И. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие/ Николаев Е.И. — Электрон.текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2015. — 225 c. — Режим доступа: http://www.iprbookshop.ru/62967.html. — ЭБС «IPRbooks».
3. Сорокин А.А. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие. Курс лекций/ Сорокин А.А. — Электрон.текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2014. — 174 c. — Режим доступа: http://www.iprbookshop.ru/63110.html. — ЭБС «IPRbooks».

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
НЕВИННОМЫССКИЙ ТЕХНОЛОГИЧЕСКИЙ ИНСТИТУТ
(ФИЛИАЛ)

Методические указания к самостоятельной работе
для студентов направления
09.03.02 «Информационные системы и технологии»
по дисциплине

«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»

Невинномысск, 2023

Методические указания разработаны в соответствии с требованиями ФГОС ВО в части содержания и уровня подготовки выпускников направления подготовки09.03.02 «Информационные системы и технологии».

Методические указания содержат рекомендации по организации самостоятельной работы студента при изучении дисциплины «Объектно-ориентированное программирование».

Составитель доцент кафедры ИСЭА Д.В. Болдырев

Ответственный редактор доцент кафедры ИСЭА А.А. Евдокимов

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4
1. ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПРИ ИЗУЧЕНИИ ДИСЦИПЛИНЫ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ» 5
1.1. Подготовка к лекциям 7
1.2. Подготовка к практическим занятиям 8
1.3. Самостоятельное изучение материала тем 10
1.4. Подготовка к экзамену 13
2. СРЕДСТВА ОЦЕНИВАНИЯ УРОВНЯ СФОРМИРОВАННОСТИ КОМПЕТЕНЦИЙ ОБУЧАЮЩИХСЯ ПРИ ИЗУЧЕНИИ ДИСЦИПЛИНЫ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ» 15
3. ОТЧЕТНОСТЬ ПО ДИСЦИПЛИНЕ 27
ЛИТЕРАТУРА 29

ВВЕДЕНИЕ

Дисциплина «Объектно-ориентированное программирование» ставит своей целью формирование следующих компетенций будущего бакалавра по направлению подготовки 09.03.02 «Информационные системы и технологии».

Код, формулировка компетенции	Код, формулировка индикатора	Планируемые результаты обучения по дисциплине (модулю), характеризующие этапы формирования компетенций, индикаторов
ПК-4 Способен разработать архитектуру ИС	ИД-$1_{ПК-4}$ Осуществляет разработку стратегии развития информационных технологий инфраструктуры предприятия и управления ее реализацией	Понимает общую методологию и технологию объектно-ориентированного программирования; выполняет объектную декомпозицию предметной области.
	ИД-$2_{ПК-4}$ Осуществляет разработку архитектуры ИТ и ИС инфраструктуры предприятия	Разрабатывает процедуры сборки модулей и компонент программного обеспечения; процедуры развертывания и обновления программного обеспечения; процедуры миграции и преобразования (конвертации) данных
	ИД-$3_{ПК-4}$ Осуществляет обоснование архитектуры ИС	Обосновывает корректность функциональной и системной архитектуры ИС; оценивает и согласовывает сроки выполнения поставленных задач.

Главными задачами дисциплины являются: формирование представлений об общей методологии и средствах технологии объектно-ориентированного программирования; углубленная подготовка студентов в области применения технологии объектно-ориентированного программирования.

В результате освоения дисциплины студент должен:

- знать принципы автоматизации решения задач организационного управления и бизнес-процессов;
- уметь выполнять работы по созданию (модификации) и сопровождению информационных систем;
- владеть навыками создания (модификации) и сопровождения информационных систем.

Методические указания предназначены для выполнения самостоятельной работе по дисциплине «Объектно-ориентированное программирование» с учетом требований ФГОС ВО для направления подготовки09.03.02 «Информационные системы и технологии». Они способствуют лучшему усвоению студентами теоретических положений и обеспечивает приобретение практических навыков по исследованию элементов и систем автоматического регулирования и управления.

1. ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПРИ ИЗУЧЕНИИ ДИСЦИПЛИНЫ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»

Самостоятельная работа студентов (далее — СРС) является неотъемлемой составляющей образовательного процесса в Университете и является обязательной для каждого студента. Основная цель СРС — освоение в полном объѐме образовательной программы и последовательное формирование компетенций эффективной самостоятельной профессиональной (практиче-

ской и научно-теоретической) деятельности. Самостоятельная работа конкретна по своей предметной направленности и сопровождается непрерывным контролем и оценкой ее результатов.

Количество часов, отводимое на самостоятельную работу, определяется учебным планом направления подготовки 09.03.02.

Содержательно самостоятельная работа студентов определяется ФГОС ВО направления подготовки 09.03.02, программой и учебно-методическим комплексом дисциплины «Объектно-ориентированное программирование».

Методика организации самостоятельной работы студентов зависит от структуры, характера и особенностей дисциплины «Объектно-ориентированное программирование», объема часов на ее изучение, вида заданий для СРС, индивидуальных возможностей студентов и условий учебной деятельности.

Формы самостоятельной работы студентов определяются содержанием дисциплины «Объектно-ориентированное программирование», степенью подготовленности студентов. Они могут быть тесно связаны с теоретическим курсом и иметь учебный или учебно-исследовательский характер. Форму самостоятельной работы студентов определяют кафедра ИСЭА при разработке программы дисциплины «Объектно-ориентированное программирование».

Самостоятельная работа может осуществляться индивидуально или группами студентов в зависимости от цели, объема, конкретной тематики самостоятельной работы, уровня сложности, уровня умений студентов.

СРС, не предусмотренная образовательной программой, учебным планом и учебно-методическими материалами, раскрывающими и конкретизирующими их содержание, осуществляется студентами инициативно, с целью реализации собственных учебных и научных интересов.

В учебном процессе выделяют аудиторную и внеаудиторную самостоятельную работу.

Аудиторная самостоятельная работа по дисциплине «Объектно-ориентированное программирование» выполняется на учебных занятиях

(лекциях, лабораторных занятиях и консультациях) под руководством преподавателя и по его заданию.

Внеаудиторная самостоятельная работа студентов выполняется во внеаудиторное время по заданию и при методическом руководстве и контроле преподавателя, но без его непосредственного участия.СРС включает в себя:

- подготовку к аудиторным занятиям (лекциям, практическим, лабораторным)и выполнение соответствующих заданий;
- работу над отдельными темами учебных дисциплин (модулей) в соответствии сучебно-тематическими планами;
- выполнение контрольной работы;
- подготовку ко всем видам промежуточных и итоговых контрольных испытаний.

1.1. Подготовка к лекциям

Главное в период подготовки к лекционным занятиям — научиться методам самостоятельного умственного труда, сознательно развивать свои творческие способности и овладевать навыками творческой работы. Для этого необходимо строго соблюдать дисциплину учебы и поведения. Четкое планирование своего рабочего времени и отдыха является необходимым условием для успешной самостоятельной работы. В основу его нужно положить рабочие программы изучаемых в семестре дисциплин.

Каждому студенту следует составлять еженедельный и семестровый планы работы, а также план на каждый рабочий день. С вечера всегда надо распределять работу на завтрашний день. В конце каждого дня целесообразно подводить итог работы: тщательно проверить, все ли выполнено по намеченному плану, не было ли каких-либо отступлений, а если были, по какой причине это произошло. Нужно осуществлять самоконтроль, который является необходимым условием успешной учебы. Если что-то осталось невы-

полненным, необходимо изыскать время для завершения этой части работы, не уменьшая объема недельного плана.

Слушание и запись лекций — сложный вид вузовской аудиторной работы. Внимательное слушание и конспектирование лекций предполагает интенсивную умственную деятельность студента. Краткие записи лекций, их конспектирование помогает усвоить учебный материал. Конспект является полезным тогда, когда записано самое существенное, основное и сделано это самим студентом. Не надо стремиться записать дословно всю лекцию. Такое «конспектирование» приносит больше вреда, чем пользы. Запись лекций рекомендуется вести по возможности собственными формулировками. Желательно запись осуществлять на одной странице, а следующую оставлять для проработки учебного материала самостоятельно в домашних условиях.

Конспект лекций лучше подразделять на пункты, параграфы, соблюдая красную строку. Этому в большой степени будут способствовать пункты плана лекции, предложенные преподавателям. Принципиальные места, определения, формулы и другое следует сопровождать замечаниями «важно», «особо важно», «хорошо запомнить» и т.п. Можно делать это и с помощью разноцветных маркеров или ручек. Лучше если они будут собственными, чтобы не приходилось просить их у однокурсников и тем самым не отвлекать их во время лекции. Целесообразно разработать собственную «маркографию» (значки, символы), сокращения слов. Не лишним будет и изучение основ стенографии. Работая над конспектом лекций, всегда необходимо использовать не только учебник, но и ту литературу, которую дополнительно рекомендовал лектор. Именно такая серьезная, кропотливая работа с лекционным материалом позволит глубоко овладеть знаниями.

1.2. Подготовка к практическим занятиям

Подготовку к каждому практическому занятию студент должен начать с ознакомления с методическими указаниями, которые включают содержание

работы. Тщательное продумывание и изучение вопросов основывается на проработке текущего материала лекции, а затем изучения обязательной и дополнительной литературы, рекомендованную к данной теме. На основе индивидуальных предпочтений студенту необходимо самостоятельно выбрать тему доклада по проблеме и по возможности подготовить по нему презентацию.

Если программой дисциплины предусмотрено выполнение практического задания, то его необходимо выполнить с учетом предложенной инструкции (устно или письменно). Все новые понятия по изучаемой теме необходимо выучить наизусть и внести в глоссарий, который целесообразно вести с самого начала изучения курса. Результат такой работы должен проявиться в способности студента свободно ответить на теоретические вопросы семинара, его выступлении и участии в коллективном обсуждении вопросов изучаемой темы, правильном выполнении практических заданий и контрольных работ.

В зависимости от содержания и количества отведенного времени на изучение каждой темы практическое занятие может состоять из четырех-пяти частей:

1. Обсуждение теоретических вопросов, определенных программой дисциплины.

2. Доклад и/или выступление с презентациями по выбранной проблеме.

3. Обсуждение выступлений по теме — дискуссия.

4. Выполнение практического задания с последующим разбором полученных результатов или обсуждение практического задания.

5. Подведение итогов занятия.

Первая часть — обсуждение теоретических вопросов — проводится в виде фронтальной беседы со всей группой и включает выборочную проверку преподавателем теоретических знаний студентов. Примерная продолжительность — до 15 минут. Вторая часть — выступление студентов с докладами, которые должны сопровождаться презентациями с целью усиления нагляд-

ности восприятия, по одному из вопросов практического занятия. Обязательный элемент доклада — представление и анализ статистических данных, обоснование социальных последствий любого экономического факта, явления или процесса. Примерная продолжительность — 20-25 минут. После докладов следует их обсуждение — дискуссия. В ходе этого этапа практического занятия могут быть заданы уточняющие вопросы к докладчикам. Примерная продолжительность — до 15-20 минут. Если программой предусмотрено выполнение практического задания в рамках конкретной темы, то преподавателями определяется его содержание и дается время на его выполнение, а затем идет обсуждение результатов. Подведением итогов заканчивается практическое занятие.

В процессе подготовки к практическим занятиям, студентам необходимо обратить особое внимание на самостоятельное изучение рекомендованной учебно-методической (а также научной и популярной) литературы. Самостоятельная работа с учебниками, учебными пособиями, научной, справочной и популярной литературой, материалами периодических изданий и Интернета, статистическими данными является наиболее эффективным методом получения знаний, позволяет значительно активизировать процесс овладения информацией, способствует более глубокому усвоению изучаемого материала, формирует у студентов свое отношение к конкретной проблеме. Более глубокому раскрытию вопросов способствует знакомство с дополнительной литературой, рекомендованной преподавателем по каждой теме семинарского или практического занятия, что позволяет студентам проявить свою индивидуальность в рамках выступления на данных занятиях, выявить широкий спектр мнений по изучаемой проблеме.

1.3. Самостоятельное изучение материала тем

Конспект — наиболее совершенная и наиболее сложная форма записи. Слово «конспект» происходит от латинского «conspectus», что означает «об-

зор, изложение». В правильно составленном конспекте обычно выделено самое основное в изучаемом тексте, сосредоточено внимание на наиболее существенном, в кратких и четких формулировках обобщены важные теоретические положения.

Конспект представляет собой относительно подробное, последовательное изложение содержания прочитанного. На первых порах целесообразно в записях ближе держаться тексту, прибегая зачастую к прямому цитированию автора. В дальнейшем, по мере выработки навыков конспектирования, записи будут носить более свободный и сжатый характер.

Конспект книги обычно ведется в тетради. В самом начале конспекта указывается фамилия автора, полное название произведения, издательство, год и место издания. При цитировании обязательная ссылка на страницу книги. Если цитата взята из собрания сочинений, то необходимо указать соответствующий том. Следует помнить, что четкая ссылка на источник — непременное правило конспектирования. Если конспектируется статья, то указывается, где и когда она была напечатана.

Конспект подразделяется на части в соответствии с заранее продуманным планом. Пункты плана записываются в тексте или на полях конспекта. Писать его рекомендуется четко и разборчиво, так как небрежная запись с течением времени становиться малопонятной для ее автора. Существует правило: конспект, составленный для себя, должен быть по возможности написан так, чтобы его легко прочитал и кто-либо другой.

Формы конспекта могут быть разными и зависят от его целевого назначения (изучение материала в целом или под определенным углом зрения, подготовка к докладу, выступлению на занятии и т.д.), а также от характера произведения (монография, статья, документ и т.п.). Если речь идет просто об изложении содержания работы, текст конспекта может быть сплошным, с выделением особо важных положений подчеркиванием или различными значками.

В случае, когда не ограничиваются переложением содержания, а фиксируют в конспекте и свои собственные суждения по данному вопросу или дополняют конспект соответствующими материалами их других источников, следует отводить место для такого рода записей. Рекомендуется разделить страницы тетради пополам по вертикали и в левой части вести конспект произведения, а в правой свои дополнительные записи, совмещая их по содержанию.

Конспектирование в большей мере, чем другие виды записей, помогает вырабатывать навыки правильного изложения в письменной форме важные теоретических и практических вопросов, умение четко их формулировать и ясно излагать своими словами.

Таким образом, составление конспекта требует вдумчивой работы, затраты времени и труда. Зато во время конспектирования приобретаются знания, создается фонд записей.

Конспект может быть текстуальным или тематическим. В текстуальном конспекте сохраняется логика и структура изучаемого произведения, а запись ведется в соответствии с расположением материала в книге. За основу тематического конспекта берется не план произведения, а содержание какой-либо темы или проблемы.

Текстуальный конспект желательно начинать после того, как вся книга прочитана и продумана, но это, к сожалению, не всегда возможно. В первую очередь необходимо составить план произведения письменно или мысленно, поскольку в соответствии с этим планом строится дальнейшая работа. Конспект включает в себя тезисы, которые составляют его основу. Но, в отличие от тезисов, конспект содержит краткую запись не только выводов, но и доказательств, вплоть до фактического материала. Иначе говоря, конспект — это расширенные тезисы, дополненные рассуждениями и доказательствами, мыслями и соображениями составителя записи.

Как правило, конспект включает в себя выписки, но в него могут войти отдельные места, цитируемые дословно, а также факты, примеры, цифры,

таблицы и схемы, взятые из книги. Следует помнить, что работа над конспектом только тогда будет творческой, когда она не ограничена текстом изучаемого произведения. Нужно дополнять конспект данными из других источников.

В конспекте необходимо выделять отдельные места текста в зависимости от их значимости. Можно пользоваться различными способами: подчеркиваниями, вопросительными и восклицательными знаками, репликами, краткими оценками, писать на полях своих конспектов слова: «важно», «очень важно», «верно», «характерно».

В конспект могут помещаться диаграммы, схемы, таблицы, которые придадут ему наглядность.

Составлению тематического конспекта предшествует тщательное изучение всей литературы, подобранной для раскрытия данной темы. Бывает, что какая-либо тема рассматривается в нескольких главах или в разных местах книги. А в конспекте весь материал, относящийся к теме, будет сосредоточен в одном месте. В плане конспекта рекомендуется делать пометки, к каким источникам (вплоть до страницы) придется обратиться для раскрытия вопросов. Тематический конспект составляется обычно для того, чтобы глубже изучить определенный вопрос, подготовиться к докладу, лекции или выступлению на семинарском занятии. Такой конспект по содержанию приближается к реферату, докладу по избранной теме, особенно если включает и собственный вклад в изучение проблемы.

1.4. Подготовка к экзамену

Экзаменационная сессия — очень тяжелый период работы для студентов и ответственный труд для преподавателей. Главная задача экзаменов — проверка качества усвоения содержания дисциплины.

На основе такой проверки оценивается учебная работа не только студентов, но и преподавателей: по результатам экзаменов можно судить и о каче-

стве всего учебного процесса. При подготовке к экзамену студенты повторяют материал курсов, которые они слушали и изучали в течение семестра, обобщают полученные знания, выделяют главное в предмете, воспроизводят общую картину для того, чтобы яснее понять связь между отдельными элементами дисциплины.

При подготовке к экзаменам основное направление дают программы курса и конспект, которые указывают, что в курсе наиболее важно. Основной материал должен прорабатываться по учебнику, поскольку конспекта недостаточно для изучения дисциплины. Учебник должен быть проработан в течение семестра, а перед экзаменом важно сосредоточить внимание на основных, наиболее сложных разделах. Подготовку по каждому разделу следует заканчивать восстановлением в памяти его краткого содержания в логической последовательности.

До экзамена обычно проводится консультация, но она не может возместить отсутствия систематической работы в течение семестра и помочь за несколько часов освоить материал, требующийся к экзамену. На консультации студент получает лишь ответы на трудные или оставшиеся неясными вопросы. Польза от консультации будет только в том случае, если студент до нее проработает весь материал. Надо учиться задавать вопросы, вырабатывать привычку пользоваться справочниками, энциклопедиями, а не быть на иждивении у преподавателей, который не всегда может тут же, «с ходу» назвать какой-либо факт, имя, событие. На экзамене нужно показать не только знание предмета, но и умение логически связно построить устный ответ.

Получив билет, надо вдуматься в поставленные вопросы для того, чтобы правильно понять их. Нередко студент отвечает не на тот вопрос, который поставлен, или в простом вопросе ищет скрытого смысла. Не поняв вопроса и не обдумав план ответа, не следует начинать писать. Конспект своего ответа надо рассматривать как план краткого сообщения на данную тему и составлять ответ нужно кратко. При этом необходимо показать умение выражать мысль четко и доходчиво.

Отвечать нужно спокойно, четко, продуманно, без торопливости, придерживаясь записи своего ответа. На экзаменах студент показывает не только свои знания, но и учится владеть собой. После ответа на билет могут следовать вопросы, которые имеют целью выяснить понимание других разделов курса, не вошедших в билет. Как правило, на них можно ответить кратко, достаточно показать знание сути вопроса. Часто студенты при ответе на дополнительные вопросы проявляют поспешность: не поняв смысла того, что у них спрашивают, начинают отвечать и нередко говорят не по сути.

Следует помнить, что необходимым условием правильного режима работы в период экзаменационной сессии является нормальный сон, поэтому подготовка к экзаменам не должна быть в ущерб сну. Установлено, что сильное эмоциональное напряжение во время экзаменов неблагоприятно отражается на нервной системе и многие студенты из-за волнений не спят ночи перед экзаменами. Обычно в сессию студенту не до болезни, так как весь организм озабочен одним — сдать экзамены. Но это еще не значит, что последствия неправильно организованного труда и чрезмерной занятости не скажутся потом. Поэтому каждый студент помнить о важности рационального распорядка рабочего дня и о своевременности снятия или уменьшения умственного напряжения.

2. СРЕДСТВА ОЦЕНИВАНИЯ УРОВНЯ СФОРМИРОВАННОСТИ КОМПЕТЕНЦИЙ ОБУЧАЮЩИХСЯ ПРИ ИЗУЧЕНИИ ДИСЦИПЛИНЫ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»

Вопросы для собеседования

Тема 1 Теоретические основы объектно-ориентированного программирования

1. Программы с глобальными и локальными данными.

2. Структурное программирование: основные принципы, пошаговая детализация, процедурная декомпозиция; достоинства и недостатки.
3. Модульное программирование: интерфейс и реализация; достоинства и недостатки.
4. Объектно-ориентированное программирование: объектная декомпозиция; достоинства и недостатки.
5. Абстрагирование.
6. Инкапсуляция (ограничение доступа).
7. Наследование (иерархичность).
8. Полиморфизм (типизация).
9. Модульность.
10. Параллелизм.
11. Устойчивость.
12. Структура класса.
13. Статические и динамические ресурсы.
14. Анализ задачи.
15. Объектная декомпозиция.
16. Логическое проектирование.
17. Физическое проектирование.
18. Эволюция системы.
19. Модификация проекта.

Тема 2 Техника объектно-ориентированного программирования

1. Статические и экземплярные ресурсы.
2. Поля и методы класса.
3. Модификаторы доступа.
4. Свойства и индексаторы, стратегии доступа.
5. Конструкторы и деструкторы.
6. Композиция, контейнерные классы.
7. Наследование: родители и потомки.

8. Полиморфизм: раннее и позднее связывание.
9. Полиморфизм на основе абстрактных классов.
10. Полиморфизм на основе виртуальных методов.
11. Перегрузка методов.
12. Перегрузка операций.
13. Делегирование методов, статическое и динамическое делегирование.
14. Параметризированные классы.
15. Простое и множественное наследование.
16. Таблица виртуальных методов.
17. Делегирование методов, статическое и динамическое делегирование.
18. Интерфейсы.
19. Обрабатывающая конструкция try … catch.
20. Финализирующая конструкция try … finally.
21. Интерфейсы.
22. Генерация исключений.
23. Порядок обработки исключений.

Критерии оценивания компетенций

Оценка «**зачтено**» выставляется студенту, если он твердо знает материал, грамотно и по существу излагает его, правильно применяет теоретические положения при решении практических вопросов и задач, владеет необходимыми навыками и приемами их выполнения. Допускаются некоторые неточности, недостаточно правильные формулировки в изложении программного материала, затруднения при выполнении практических работ.

Оценка «**не зачтено**» выставляется студенту, если он не знает значительной части программного материала, допускает существенные ошибки, неуверенно, с большими затруднениями выполняет практические задания.

Вопросы к экзамену

1. Эволюция технологии разработки программных продуктов
2. Объектная декомпозиция
3. Разработка программ с использованием объектно-ориентированной технологии
4. Основные принципы объектно-ориентированного программирования
5. Классы и объекты
6. Организация класса. Модификаторы доступа. Статические и экземплярные ресурсы.
7. Поля и методы класса.
8. Конструкторы и деструкторы.
9. Композиция. Контейнерные классы.
10. Наследование: родители и потомки.
11. Простое и множественное наследование.
12. Полиморфизм: раннее и позднее связывание.
13. Полиморфизм на основе абстрактных классов.
14. Полиморфизм на основе виртуальных методов.
15. Статическое делегирование методов.
16. Динамическое делегирование методов.
17. Параметризированные классы.
18. Интерфейсы.
19. Генерация исключений. Порядок обработки исключений.
20. Обрабатывающая конструкция try … catch.
21. Финализирующая конструкция try … finally.

Критерии оценивания компетенций

Оценка «**отлично**» выставляется студенту, если он глубоко и прочно усвоил программный материал, исчерпывающе, последовательно, четко и логически стройно его излагает, умеет тесно увязывать теорию с практикой, свободно справляется с задачами, вопросами и другими видами применения

знаний, причем не затрудняется с ответом при видоизменении заданий, использует в ответе материал монографической литературы, правильно обосновывает принятое решение, владеет разносторонними навыками и приемами выполнения практических задач.

Оценка «**хорошо**» выставляется студенту, если он твердо знает материал, грамотно и по существу излагает его, не допуская существенных неточностей в ответе на вопрос, правильно применяет теоретические положения при решении практических вопросов и задач, владеет необходимыми навыками и приемами их выполнения.

Оценка «**удовлетворительно**» выставляется студенту, если он имеет знания только основного материала, но не усвоил его деталей, допускает неточности, недостаточно правильные формулировки, нарушения логической последовательности в изложении программного материала, испытывает затруднения при выполнении практических работ.

Оценка «**неудовлетворительно**» выставляется студенту, который не знает значительной части программного материала, допускает существенные ошибки, неуверенно, с большими затруднениями выполняет практические работы.

Компетентностно-ориентированные задания и задачи

ЗАДАЧА 1

Цель:

Ознакомление с концепциями инкапсуляции и модульности. Изучение приемов работы с классами, конструкторами и деструкторами, разработка интерфейса методов класса, создание и работа с экземплярами класса. Освоение принципа «класс-элемент — класс-набор».

Задание:

Дан объект согласно вариантам, который является элементом для набора. Элемент состоит из компонент, которые хранятся в нем. Над элементом определены операции:

- получение значения компоненты элемента;
- установка и инициализация значения компоненты элемента;
- контроль значения компоненты элемента (на допустимый диапазон);
- копирование элемента.

Из элементов строится набор. Элементы в наборе проиндексированы от стартового значения. Размер набора задается при создании. Над набором определены операции:

- установка стартового индекса, получение диапазона индексов;
- заполнение набора случайными значениями;
- получение и изменение элемента набора по индексу;
- сортировка элементов по возрастанию и по убыванию;
- дополнительные операции согласно вариантам.

Необходимо разработать:

- класс для описания элемента и его свойств;
- класс для описания набора и его свойств;
- методы работы с элементом и с набором для перечисленных операций;
- дефолтный, копирующий, параметрический конструкторы для создания экземпляров набора и экземпляров элемента;
- интерфейс для редактирования элемента и интерфейс для редактирования набора, отображения и изменения их свойств.

Требования:

Интерфейс и все классы реализуются в одном модуле. Для редактирования элемента разработать функцию ModifyElement(), которая должна получать ссылку на экземпляр элемента и предоставлять интерактивный консольный интерфейс для работы с ним. Для редактирования набора разработать

функцию ModifyPalette(), которая должна получать ссылку на экземпляр набора и предоставлять интерактивный консольный интерфейс для работы с ним. Функция Main() запрашивает количество элементов, создает экземпляр набора параметрическим конструктором и вызывает функцию ModifyPalette(), которая использует ModifyElement().

Комментарии:

Набор хранит элементы как динамический массив, определяя его размер при создании. При копировании набора, копируются все его элементы.

Варианты заданий:

0. Элемент: цвет в формате CYMK. Дополнительно: сортировка по компонентам и целиком, пересчет в RGB.

1. Элемент: положение солнца в координатах a-азимут, z-зенит, ^-горизонт. Дополнительно: сортировка по компонентам и целиком, пересчет в другую систему координат (на выбор).

2. Элемент: цвет в формате YUV. Дополнительно: сортировка по компонентам и целиком, пересчет в RGB.

3. Элемент: положение солнца в экваториальных координатах - склонение, *p* — полярное расстояние, *t*— часовой угол. Дополнительно: сортировка по компонентам и целиком, пересчет в другую систему координат (на выбор).

4. Элемент: цвет в формате AHSL. Дополнительно: сортировка по компонентам и целиком, пересчет в RGB.

5. Элемент: цвет в формате RYB. Дополнительно: сортировка по компонентам и целиком, пересчет в RGB.

6. Элемент: декартовы координаты в пространстве (x, y) — координаты. Дополнительно: сортировка по компонентам и целиком, пересчет в цилиндрическую систему координат.

7. Элемент: цвет в формате YIQ. Дополнительно: сортировка по компонентам и целиком, пересчет в RGB.

8. Элемент: время в формате h — часы, m — минуты, s — секунды. До-

полнительно: сортировка по компонентам и целиком, пересчет в 12-ти часовой формат времени (AMи PM).

9. Элемент: цвет в формате HSV. Дополнительно: сортировка по компонентам и целиком, пересчет в RGB.

ЗАДАЧА 2

Цель:

Ознакомление с концепциями полиморфизма и типизации. Изучение перегрузки операторов. Разработка классов с перегруженными операторами и программирование выражений с их помощью.

Задание:

Дан объект данных, над которым определены операции согласно вариантам. Реализовать набор операций для работы с объектом так, чтобы его можно было использовать в выражениях, не прибегая к вызову функций. Необходимо разработать:

- класс объекта и определить правила выполнения операций над ним;
- набор перегруженных операторов, реализующих операции с объектом;
- интерфейс для редактирования объекта с помощью операторов;
- интерфейс для тестирования использования объекта в выражениях.

Требования:

Интерфейс и класс объекта реализуются в одном модуле. Для редактирования объекта разработать функцию ModifyObject(), которая должна получать ссылку на экземпляр объекта и предоставлять интерактивный консольный интерфейс для работы с ним. Для тестирования использования объекта в выражениях разработать функцию Main(), которая должна предоставлять интерактивный консольный интерфейс, демонстрирующий применение объекта в выражениях.

Обязательной реализации подлежат следующие операции: сложение (+ и +=), вычитание (- и -=), умножение (* и *=), сравнение на равенство (== и !=), унарные (+, -), инверсия (~), присваивание (=), проверка на ноль (!), преобразование к типу (type), ввод из потока (cin>>) и вывод в поток (cout<<). Разработать набор тестовых примеров, демонстрирующих использование перегруженных операторов в арифметических и логических выражениях.

Комментарии:

При перегрузке операторов необходимо учитывать:

- нельзя определить новый лексический символ для оператора;
- нельзя изменить приоритет операторов;
- нельзя изменить арность операторов;
- нельзя перегрузить оператор для стандартных типов данных;
- нельзя перегружать не перегружаемые операторы (.), (::), (?_:_), (sizeof), (typeid), (,);
- некоторые операторы можно перегружать только как методы класса (=), [ ], ( ), (–>);
- если оператор можно использовать и как унарный, и как бинарный, например, (&), (*), (+), (–), то каждый способ применения перегружается отдельно;
- перегруженные операторы не могут иметь аргументов по умолчанию;
- перегруженные операторы должны учитывать смысловую эквивалентность: var = var + 1; var += 1; var++; ++var; var –= (–1);
- поведение перегруженных операторов должно соответствовать их смысловому содержанию для определяемых типов данных.

Варианты заданий:

0. Объект: комплексное число (вещественная и мнимая части). Принять: (!) — проверка на ноль, (+ и +=) — сложение, (- и -=) — вычитание, (* и *=) — умножение, (== и !=) — сравнение, (double) — вычисление модуля,

(float) — вычисление аргумента, (~) — сопряженное число.

1. Объект: интервал времени (часы, минуты, секунды). Реализовать операции с учетом ограничений на часы (0 до 23), минуты и секунды (0 до 59), т.е. результат всегда от 0:0:0 до 23:59:59. Принять: (+ и +=) — сложение, (- и -=) — вычитание, (* и *=) — удлинение или сокращение, (!) — проверка на ноль, (== и !=) — сравнение, (long) — преобразование в секунды, (float) — преобразование в часы (3600 сек), (~) — дополнение до конца суток.

2. Объект: денежная сумма (признак валюты [р., $], сумма в номинале [рубли, доллары], сумма в размене [копейки, центы]). Реализовать операции с учетом конвертации, если валюты не совпадают. Принять: (+ и +=) — сложение, (- и -=) — вычитание, (* и *=) — умножение, (!) — проверка на ноль, (== и !=) — сравнение, (float) — в номинал, (int) — в размен, (~) — изменение признака валюты с конвертацией, (%) — процент от суммы.

3. Объект: интервал даты (часов, дней, лет). Реализовать операции с учетом столетия (0 до 99) и ограничений на дни (0 до 364) и часы (0 до 23), т.е. результат всегда от 0-0-0 до 23-364-99. Принять: (* и *=) — удлинение или сокращение, (+ и +=) — сложение, (- и -=) — вычитание, (== и !=) — сравнение, (!) — проверка на ноль, (long) — преобразование в часы, (float) — преобразование в года (365 дней), (~) — дополнение до конца столетия.

4. Объект: расстояние (сажень, аршин, вершок). 1 сажень = 3 аршинам, 1 аршин = 16 вершкам, 1 вершок = 44,5 мм. Результат всегда от 0 до 500 саженей (1 верста). Принять: (+ и +=) — сложение, (- и -=) — разность, (* и *=) — удлинение или сокращение, (== и !=) — сравнение, (!) — проверка на ноль, (double) — преобразование в миллиметры, (int) — преобразование в вершки, (~) — дополнение до версты (500 саженей).

5. Объект: строка символов (0 до 128). Принять: (+ и +=) — соединение строк, повторение символа, (- и -=) — отсечение строки, (*) — поиск подстроки, (*=) — заполнение подстрокой или символом, (== и !=) — сравнение, (!) — проверка на пусто, (~) — переворот наоборот, (int) — длина строки.

6. Объект: натуральная дробь (целое, числитель, знаменатель). Реали-

зовать операции с учетом приведения к общему знаменателю. Принять: (+ и +=) — сложение, (- и -=) — вычитание, (* и *=) — умножение, (!) — проверка на ноль, (== и !=) — сравнение, (double) — преобразование в рациональную дробь, (~) — взаимообратная натуральная дробь.

7. Объект: угол (градусы, минуты, секунды). Реализовать операции с учетом целых оборотов и ограничений на градусы (0 до 359), минуты и секунды (0 до 59), т.е. результат всегда от 0°0'0" до 359°59'59". Принять: (+ и +=) — сложение, (- и -=) — вычитание, (* и *=) — умножение, (!) — проверка на ноль, (== и !=) — сравнение, (double) — преобразование в радианы, (int) — преобразование в секунды, (~) — обратный угол до 360°.

8. Объект: квадратная матрица [3x3]. Реализовать операции над матрицами. Принять: (+ и +=) — сложение, (- и -=) — вычитание, (* и *=) — умножение, (!) — проверка на ноль, (== и !=) — сравнение, (double) — вычисление детерминанта, (int) — количество ячеек, (~) — транспонирование.

9. Объект: алфавит (только прописные от Адо Z). Реализовать операции над алфавитами как над множествами. Принять: (+ и +=) — объединение, (!) — проверка на пусто, (- и -=) — разность, (* и *=) — пересечение, (== и !=) — сравнение, (int) — количество букв, (~) — отрицание как замена на буквы, которых нет.

ЗАДАЧА 3

Цель:

Ознакомление с концепциями наследования и абстракции. Обычное и множественное наследование классов. Статические свойства, методы, классы. Перегрузка методов при наследовании классов. Освоение приемов создания, редактирования, копирования, удаления экземпляров и разработки интерфейсов классов.

Задание:

Дана фигура на плоскости согласно вариантам. Фигура описывается индивидуальными геометрическими свойствами и общими оформительскими

свойствами: цвет, видимость. У фигуры имеются характеристики: периметр, площадь, ограничивающая область. Область размещения фигур в плоскости ограничена экстентами, за которые фигура не должна выходить.

Необходимо разработать:

- классы для описания положения «Location» и ограничивающей области «Clip» в плоскости;
- статический класс «Geometry» для хранения общих констант и методов проверки различных ограничений на размещение фигур в плоскости;
- класс геометрического примитива «Primitive» для хранения и редактирования оформительских свойств фигуры как наследника от статического класса «Geometry»;
- класс примитивной фигуры — точки «Point» как наследника от классов «Location» и «Primitive»;
- класс фигуры согласно варианту «Figure» как наследника от класса «Point» с описанием специфических свойств и методов фигуры;
- наборы конструкторов для создания экземпляров каждого класса различными способами (дефолтный, копирующий, параметрический);
- методы для изменения свойств и вычисления характеристик фигуры;
- интерфейс для отображения и изменения всех свойств фигуры.

Требования:

Интерфейс и классы реализуются в одном модуле. Для редактирования фигуры разработать функцию ModifyFigure(), которая должна получать ссылку на экземпляр фигуры и предоставлять интерактивный консольный интерфейс для работы с ним.

Варианты заданий:

0. Фигура: сектор окружности.
1. Фигура: треугольник Рело.
2. Фигура: правильный шестиугольник.
3. Фигура: эллипс.

4. Фигура: параллелограмм.
5. Фигура: сегмент окружности.
6. Фигура: ромб.
7. Фигура: кольцо (бублик).
8. Фигура: правильная трапеция.
9. Фигура: дельтоид.

3. ОТЧЕТНОСТЬ ПО ДИСЦИПЛИНЕ

В рамках рейтинговой системы успеваемость студентов по дисциплине оценивается в ходе текущего контроля успеваемости и промежуточной аттестации.

Максимально возможный балл за весь текущий контроль устанавливается равным 55. Текущее контрольное мероприятие считается сданным, если студент получил за него не менее 60% от установленного для этого контроля максимального балла. Рейтинговый балл, выставляемый студенту за текущее контрольное мероприятие, сданное студентом в установленные графиком контрольных мероприятий сроки, определяется следующим образом:

Уровень выполнения контрольного задания	Рейтинговый балл (в % от максимального балла за контрольное задание)
Отличный	100
Хороший	80
Удовлетворительный	60
Неудвлетворительный	0

Промежуточная аттестация

Промежуточная аттестация в форме экзамена предусматривает проведение обязательной экзаменационной процедуры и оценивается 40 баллами из 100. Минимальное количество баллов, необходимое для допуска к экзамену, составляет 33 балла. Положительный ответ студента на экзамене оценивается

рейтинговыми баллами в диапазоне от **20** до **40** (**20**$\leq S_{экз} \leq$**40**), оценка меньше 20баллов считается неудовлетворительной.

Шкала соответствия рейтингового балла экзамена 5-балльной системе

Рейтинговый балл по дисциплине	Оценка по 5-балльной системе
35-40	Отлично
28-34	Хорошо
20-27	Удовлетворительно

Итоговая оценка по дисциплине, изучаемой в одном семестре, определяется по сумме баллов, набранных за работу в течение семестра, и баллов, полученных при сдаче экзамена:

Шкала пересчета рейтингового балла по дисциплине в оценку по 5-балльной системе

Рейтинговый балл по дисциплине	Оценка по 5-балльной системе
88-100	Отлично
72-87	Хорошо
53-71	Удовлетворительно
менее 53	Неудовлетворительно

Для студентов заочной формы обучения рейтинговая оценка знаний не предусмотрена

ЛИТЕРАТУРА

Перечень основной литературы:

1. Иванова Г.С. Объектно-ориентированное программирование [Электронный ресурс]: учебник/ Иванова Г.С., Ничушкина Т.Н. — Электрон. текстовые данные. — Москва: Московский государственный технический университет имени Н.Э. Баумана, 2014. — 456 с. — Режим доступа: http://www.iprbookshop.ru/94030.html. — ЭБС «IPRbooks».
2. Маляров А.Н. Объектно-ориентированное программирование [Электронный ресурс]: учебник для технических вузов/ Маляров А.Н. — Электрон. текстовые данные. — Самара: Самарский государственный технический университет, ЭБС АСВ, 2017. — 332 с. — Режим доступа: http://www.iprbookshop.ru/91772.html.— ЭБС «IPRbooks».
3. Мурадханов С.Э. Информатика и программирование: объектно-ориентированное программирование (на основе языка C#) [Электронный ресурс]: учебник/ Мурадханов С.Э., Широков А.И. — Электрон. текстовые данные. — Москва: Издательский Дом МИСиС, 2015. — 309 с. — Режим доступа: http://www.iprbookshop.ru/98855.html.— ЭБС «IPRbooks».

Перечень дополнительной литературы:

1. Зыков С.В. Введение в теорию программирования. Объектно-ориентированный подход [Электронный ресурс]: учебное пособие/ Зыков С.В. — Электрон. текстовые данные. — Москва: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021. — 187 с. — Режим доступа: http://www.iprbookshop.ru/102007.html.— ЭБС «IPRbooks».
2. Николаев Е.И. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие/ Николаев Е.И. — Электрон. текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2015. — 225 с. — Режим доступа: http://www.iprbookshop.ru/62967.html. — ЭБС «IPRbooks».

3. Сорокин А.А. Объектно-ориентированное программирование [Электронный ресурс]: учебное пособие. Курс лекций/ Сорокин А.А. — Электрон. текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2014. — 174 c. — Режим доступа: http://www.iprbookshop.ru/63110.html. — ЭБС «IPRbooks».