**Владимирский государственный университет**

**М. В. ШИШКИНА**

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

**Практикум**

**Владимир 2020**

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

М. В. ШИШКИНА

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

## Практикум

*Электронное издание*


Владимир 2020

ISBN 978-5-9984-1201-1

УДК 004.42
ББК 32.973

Рецензенты:

Кандидат технических наук
генеральный директор ООО «ФС Сервис»
*Д. С. Квасов*

Кандидат физико-математических наук
зав. кафедрой функционального анализа и его приложений
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
*В. Д. Бурков*

**Шишкина, М. В.** Объектно-ориентированное программирование [Электронный ресурс] : практикум / М. В. Шишкина ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2020. – 104 с. – ISBN 978-5-9984-1201-1. – Электрон. дан. (1,80 Мб). – 1 электрон. опт. диск (CD-ROM). – Систем. требования: Intel от 1,3 ГГц ; Windows XP/7/8/10 ; Adobe Reader ; дисковод CD-ROM. – Загл. с титул. экрана.

В издании показаны теоретические основы, разобрано достаточное количество примеров, приведены задания по лабораторным работам, задания для самостоятельной работы и контрольные вопросы для самоконтроля и закрепления материала.

Предназначен для студентов бакалавриата, обучающихся по направлению 02.03.01 «Математика и компьютерные науки».

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Ил. 26. Табл. 1. Библиогр.: 6 назв.

УДК 004.42
ББК 32.973

ISBN 978-5-9984-1201-1

# ОГЛАВЛЕНИЕ

# ПРЕДИСЛОВИЕ

Стремительное развитие информационной индустрии и проникновеннее её во все сферы современной жизни привело к усложнению временных затрат на разработку, сопровождение и модификацию программ. Концепция объектно-ориентированного программирования появилась в ответ на возникшие сложности при реализации поставленных перед программистами задач методами структурного программирования.

В настоящие время объектно-ориентированный подход проник во многие языки программирования. Овладение навыками объектно-ориентированного программирования дает специалисту мощный инструмент в создании крайне востребованных в современном мире визуальных приложений. Умение описать на языке программирования объекты бытовой и профессиональной деятельности позволяет сократить программный код, сделать его более наглядным и читаемым, тем самым сокращая время на разработку и модификацию, дает возможность для работы в команде.

Практикум содержит задания для лабораторных работ, самостоятельной работы и вопросы для самоконтроля, всё это входит в состав учебно-методического комплекса дисциплины, разработанного на основе современных федеральных государственных образовательных стандартов подготовки бакалавров по направлению 02.03.01 «Математика и компьютерные науки».

Практикум разработан в соответствии с рабочей программой дисциплины «Объектно-ориентированное программирование» и может быть рекомендован в качестве основного пособия для выполнения лабораторных работ и самостоятельной работы при освоении дисциплины.

# ВВЕДЕНИЕ

Практикум состоит из двух больших взаимосвязанных частей. Первая часть практикума основана на изученных студентами ранее основах программирования и базируется на знаниях, основ языка программирования высокого уровня *C++*, принципов структурного и модульного программирования, основных алгоритмических структур, умении реализовывать на языке программирования высокого уровня комбинированных алгоритмов, представленных в виде словесного описания, математической формулы, блок-схемы.

Первая часть курса позволяет студентам получить представление о парадигме объектно-ориентированного программирования. Знакомит обучающихся с основными принципами объектно-ориентированного программирования.

Второй раздел курса позволяет студентам закрепить и углубить свои знания и навыки в области объектно-ориентированного программирования, расширить представления о языках программирования высокого уровня, получить знания об основах разработки визуальных приложений.

Структура практикума организована таким образом, что каждая следующая работа основана на знаниях и навыках, полученных при выполнении предыдущей работы. Все лабораторные задания предваряются подробной теоретической информацией, содержащей достаточное количество примеров, представлены фрагменты кода и иллюстрации работы программы, позволяющие лучше понять суть и процесс выполняемых действий.

После каждого задания для лабораторной работы следует перечень вопросов, помогающих студенту самостоятельно оценить уровень своих знаний и закрепить их. Ответы на поставленные вопросы прямо или косвенно содержатся в материале, предваряющем задания к лабораторным работам. Каждая тема содержит задания для самостоятельной работы, позволяющие закрепить навыки, полученные во время выполнения работы.

В приложении практикума содержатся требования к отчету по работе, рекомендации по форматированию отчёта, краткая справочная информация по языкам программирования C++ и C#.

# ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

## Основы объектно-ориентированного программирования

### *Введение в объектно-ориентированного программирование*

Программный код, написанный по принципам объектно-ориентированного программирования представляет собой набор абстрактных сущностей, обладающих состоянием и поведением, а также и способов их взаимодействия.

В программировании под *классом* понимают пользовательский тип, объединяющий в себе разнотипные данные, называемые полями или атрибутами, и функции, обрабатывающие эти данные. Такая функция носит название *метод* класса и определяет поведение, т.е. способ взаимодействия между экземпляром и другими сущностями, реакции на внешние воздействия экземпляров класса (переменных соответствующего типа).

Для описания класса необходимо указать ключевое слово `class`, далее имя класса, после чего в фигурных скобках описать все поля данные и методы этого класса.

```
class <имя касса>
{описание полей и методов класса;};
```

В качестве примера опишем класс `Kvadrat`, имеющий одно вещественное поле, хранящее сторону квадрата и метод вычисляющий и возвращающий площадь квадрата, `Sq()`.

```
class Kvadrat
{   float a;// поле – сторона квадрата;
public:
    Kvadrat(float A) { a = A; };
    Kvadrat() { a = 10; };
    float Get_a () { return a; };
    void Set_a(float A){ a = A; };
    float Sq() { return a*a; };
};
```

Переменные класса, называют экземплярами класса или *объектами* соответствующего класса.

Объединение данных, описывающих одну сущность и методов для обработки этих данных называют *инкапсуляцией*. Инкапсуляция – основное свойство ООП.

Данные объекта должны быть скрыты от внешнего использования. Для этой цели служат *модификаторы доступа*. Модификаторы доступа – это ключевые слова, после указания которых данные и методы становятся доступны или закрыты для внешнего, т.е. за пределами класса, использования.

Модификатор доступа `private` запрещает доступ к данным класса за его пределами, `public` – открывает.

Принято поля-данные задавать с модификатором `private`, иначе происходит нарушение принципа инкапсуляции. Методы же наоборот задают с модификатором `public`, иначе будет отсутствовать возможность вызова метода за пределами класса.

При необходимости доступа к полям, получения или изменения их значений используют `Get, Set` методы. О которых будет сказано позднее.

Модификатор доступа `protected` используют, в случае, когда необходимо защитить данные от внешнего использования, но планируется передать эти данные классам наследникам. Механизм наследования будет рассмотрен далее.

По умолчанию используется модификатор доступа `private`. Именно таким образом, в рассмотренном выше примере, поле класса объявлено с модификатором доступа `private`, несмотря на то, что явно этот модификатор не указан.

Если не указать модификатор доступа `public` явно, то никакие методы нельзя будет вызвать вне класса для его объектов. В том числе и создать объект класса с заданными параметрами. Т.к. при его создании вызывается специфический метод класса, называемый конструктором.

### *Конструкторы*

*Конструктор* - это особый метод класса, вызываемый каждый раз при создании объектов этого класса и инициализирующий значения его полей.

Конструктор имеет тоже самое имя, что и класс. В заголовочной строке конструктора перед его именем тип возвращаемого значения не указывается, даже `void`.

Объявление конструктора может быть, например, таким.

`Kvadrat() { … }; или Kvadrat(float A) { a = A; };`

При необходимости можно описать несколько конструкторов в одном классе. В этом случае будет задействован механизм перегрузки функций, т.е. у всех конструкторов будет одинаковое имя, совпадающее с именем класса, и разные списки формальных параметров. Если конструктор не имеет параметров, то он носит название *конструктор по умолчанию*, возможно создание конструктора с параметрами по умолчанию.

Если программистом явно не описал, ни один конструктор, то будет автоматически создан конструктор по умолчанию, инициализирующий поля класса значениями по умолчанию для соответствующих типов данных.

Ниже приведён пример класса `Kvadrat.` Описанный класс включает в себя следующие элементы.

– Одно вещественное поле, описывающее сторону квадрата.

– Два конструктора, конструктор с одним вещественным параметром и конструктор по умолчанию, задающий сторону квадрата равной десяти.

– Метод вычисления площади квадрата. Метод возвращает вычисленное значение в точку вызова, где оно может быть использовано по усмотрению программиста. При этом само значение площади в объекте класса сохранено не будет, его просто негде хранить. Т.к. в классе есть только одно поле, хранящее сторону квадрата.

– Метод `Set()`, позволяющий задать значение стороны квадрата, т.е. значение поля *a*.

– Метод `Get(),` возвращающий это значение в точку вызова этого метода.

```
class Kvadrat
{float a;
public:
    Kvadrat(float A) { a = A; };
    Kvadrat() { a = 10; };
    float Get_a () { return a; };
    void Set_a(float A){ a = A; };
```

```
    float Sq() { return a*a; };
};// Kvadrat
```

Т.к. объект можно рассматривать как переменную соответствующего класса, то и действовать при его создании нужно как при объявлении переменной.

Т.е. указать имя типа (имя класса) и имя переменной (имя объекта). При этом, если планируется вызов конструктора с параметрами, то их нужно указать в круглых скобках после имени объекта.

```
<имя класса> <имя объекта>;
```

или

```
<имя класса> <имя объекта> (<фактические параметры>);
```

Например, создание объектов описанного выше класса должно выглядеть так.

```
Kvadrat kv1;
Kvadrat kv2(4);
```

При этом в памяти будет отведено место под поля данные, методы не дублируются, они описаны один раз в классе.

Для того что бы вызвать метод для обработки полей какого-либо объекта, необходимо указать его (метода) имя после имени объекта, для которого он вызывается, разделив их точкой. После имени метода обязательны круглые скобки, так как метод - это функция. Скобки могут быть пустыми или содержать фактические параметры, если они требуются.

```
<имя объекта>.<имя метода>([фактические параметры]);
```

Если метод возвращает значение, оно может быть использовано в точке вызова также как возвращаемое значение любой функции.

Пример вызова метода, описанного выше класса.

```
Kvadrat1.Set_a(kv2.Get_a());
Kvadrat2.Set_a(4.5);
printf("S1=%f   S2=%f\n\n", kv1.Sq(), kv2.Sq());
```

В качестве параметра метода может быть передан объект того же или другого класса, например, если необходимо вычислить сумму одноимённых полей двух объектов.

```
float Sum (Kvadrat ob)
{return Sq()+ ob.Sq()}
```

### *Указатель this*

Объект, для которого вызван метод, передавать в метод не нужно. Он будет передан неявно, через указатель `this.`

Указатель `this` имеет тип - указатель на объект класса. В момент вызова метода этот указатель получает адрес объекта, для которого метод вызывается. К указателю `this` открыт доступ пользователю, т.е. *this* можно использовать в теле метода.

Например, если необходимо вернуть из метода сам объект, то следует писать `*this` после `return`.

```
return *this;
```

В теле метода обращение к полям объекта, для которого вызван метод, возможно на прямую, без указания имени этого объекта. Так как при описании метода нет информации, для какого объекта он будет вызван, то информация об этом объекте передаётся через его адрес в скрытом параметре *this*. Вот пример такого обращения.

```
void Set_a(float A){ a = A; };
```

При этом компилятором будет сгенерировано обращение к полям соответствующего объекта через указатель `this.`

```
this-> a = A;
```

Здесь операция стрелочка `->` заменяет совокупность операций звёздочка, точка, т.е. разыменовывание и обращение к полю.

### *Деструкторы*

*Деструктором* называют особый метод класса, предназначенный для освобождения памяти, занимаемой объектом.

Деструктор вызывается каждый раз при выходе из области видимости объекта и при вызове операции `delete` для динамических объектов.

Имя деструктора начинается с символа тильда (~), далее следует имя класса.

– Деструктор не имеет аргументов и возвращаемого значения.

– Деструктор не наследуется.

– Если деструктор не определён явным образом, компилятор автоматически создают пустой деструктор.

– Нельзя объявить указатель на деструктор.

– Деструктор описывают явным образом, при необходимости выполнить какие-то дополнительные действия перед освобождением памяти, занимаемой объектом.

Например, если объект содержит поля, настроенные на динамически выделяемую память, если в этом случае не освободить эту память явно, то она будет считаться занятой даже после уничтожения объекта.

Пример описания деструктора.

```
~ classname () {delete p;}
```

### *Дружественные функции*

*Дружественной классу функцией* называют внешнюю функцию, которой открыт доступ к скрытым членам класса.

Для того что бы функция была дружественной её необходимо объявить в классе с ключевым словом `friend`.

```
void friend print(Kvadrat ob);
```

Так как дружественная функция - это внешняя функция, ей не передаётся указатель `this`, а значит объект в такую функцию нужно передавать явно.

### *Перегрузка операций*

В зыке *C++* существует возможность перегрузки большинства операций. При этом сохраняется возможность использовать их в привычном операторном виде.

Запрещена перегрузка следующих операций: ., *., ?:, #, ##, `sizeof()`. Перегрузка остальных операций разрешена.

Для того чтобы перегрузить оператор необходимо указать в заголовке функции ключевое слово `operator` и сразу за ним, знак операции. При этом обязательно сохранение количества аргументов. Приоритет операции и правила ассоциации при перегрузке сохраняются.

Перегружать операторы можно только для пользовательских типов, для базовых - нельзя.

Перегруженные функции-операции не могут иметь аргументов по умолчанию.

Такие методы наследуются, исключение составляет операция равно.

Такие методы не могут определяться как `static`.

Перегрузить оператор можно как метод класса, можно как дружественную функцию, можно как внешнюю функцию.

***Перегрузка бинарных операторных функций***

При перегрузке бинарной операции как метода, необходимо объявить метод нестатическим членом класса. Этот метод должен иметь один явный параметр – правый операнд перегружаемой операции. Левый операнд передаётся неявно через указатель `this`. Это адрес объекта, вызвавшего метод.

Если бинарная функция перегружается как внешняя функция, то оба операнда необходимо передавать явно. Т.к. такая функция не имеет неявного параметра `this`.

Перегрузим бинарный оператор + для класса *A*, описанного следующим образом.

```
class A {
protected:
int a;
public: A() { a = 2; }; A(int a1) { a = a1; };
void print() { printf("a=%i \n", a); }
};
```

Оператор будет иметь один явный формальный параметр – объект класса *A*. Который будет выступать в качестве второго слагаемого.

Первое слагаемое будет передано неявно через указатель *this*.

Метод будет возвращать новый объект класса *A*, со значением поля равным сумме полей объектов слагаемых.

```
A operator + (A ob) {return A(a+ob.a);}
```

Вызовем перегруженный оператор для объектов класса *A*.

```
A ob_A1(1);
A ob_A2(2);
A ob_A3=ob_A1+ ob_A2;
ob_A3.print();
```

На рисунке 1 приведён результат работы фрагмента кода, представленного выше.

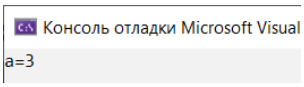



Рис. 1. Результат работы фрагмента программы

Операторные функции перегружают, с целью использования их при вызове в привычном операторном виде.

Однако перегруженный нами оператор является функцией членом класса *A*, т.е. – методом, а значит может быть вызван по правилам вызова метода. Для этого необходимо поставить точку после первого слагаемого, далее указать полное имя оператора *operator+* и передать в круглых скобках фактический параметр - второе слагаемое.

```
A ob_A4 = ob_A1.operator+(ob_A2);
ob_A4.print();
```

Результат работы, фрагмента кода представлен на рисунке 2. Полученный результат совпадает с результатом работы предыдущего фрагмента кода. Таким образом мы продемонстрировали две возможности вызова перегруженного оператора, приводящие к одному результату.

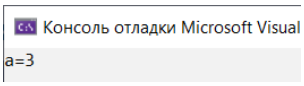



Рис. 2. Результат работы фрагмента программы

Опишем класс *B* и перегрузим в нём оператор + как дружественную функцию. Т.к. дружественная функция не имеет указателя `this`, ей необходимо передать два параметра – первое и второе слагаемое явно.

```
class B
{ protected: int b;
public: B() { b = 3; };
B(int b1) { b = b1; };
void print() { printf("b=%i \n", b); }
int friend operator+(B ob1, B ob2) { return ob1.b +
ob2.b; }
};// класс B
```

Описанная дружественная функция возвращает не объект класса, как в предыдущем примере, а целое число – сумму полей слагаемых.

Вызовем описанную дружественную функцию.

```
B b1(3); B b2(4);
int S = b1 + b2;
printf("b1 + b2=%i\n",S);
```

Результат работы приведённого выше фрагмента кода, представлен на рисунке 3.

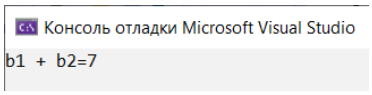



Рис. 3. Результат работы программы

***Перегрузка унарных операторных функций***

Если унарная функция перегружается как метод, то этот метод должен быть не статическим и не должен явно принимать параметров. Т.к. унарная операция имеет один операнд и это объект для которого будет вызван метод. Объект будет передан не явно в метод через указатель `this`.

Если унарный метод перегружается как внешняя функция, то такая функция должна иметь один передаваемый параметр – объект, поля которого будет обрабатывать функция.

При перегрузке постфиксной и префиксной функций необходимо учесть, их отличие в возвращаемом значении.

Т.к. префиксная функция возвращает в точку вызова новое (измененное значение), то необходимо вернуть из функции объект, предварительно изменив его значение.

Перегрузим префиксный инкремент в описанном выше классе *A* как метод класса, а префиксный декремент как дружественную классу *A* функцию.

В первом случае т.к. это метод класса явно параметр передавать не нужно.

Во втором, при перегрузке в качестве дружественной функции объект нужно передать явно в качестве параметра. Кроме того, объект должен быть передан по ссылке, иначе будет изменена только его локальная копия. В точку вызова будет возвращено новое значение, а в дальнейших операциях использовано старое значение.

```
class A {
protected: int a;
public: A() { a = 2; };
A(int a1) { a = a1; };
void print() { printf("a=%i \n", a); };
A operator ++() { ++a; return *this; }
A friend operator--(A &ob) { ob.a--; return ob; }
};// class A
```

Создадим объект класса *A* и продемонстрируем на нём работу написанных функций.

```
A ob_A1(1);
printf("ob_A1\t");
ob_A1.print();
printf("++ob_A1\t"); (++ob_A1).print();
printf("ob_A1\t"); ob_A1.print();
printf("--ob_A1\t"); (--ob_A1).print();
printf("ob_A1\t");    ob_A1.print();
```

Результат работы фрагмента кода представлены на рисунке 4.

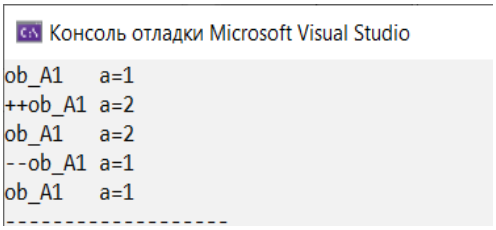



Рис. 4. Результат вызова перегруженных функций

***Задание для самостоятельной работы***

*Передайте в качестве параметра дружественной функции объект по значению и убедитесь в том, что значение внешнего объекта не будет изменено.*

При перегрузке постфиксного инкремента вводят дополнительный фиктивный целочисленный параметр, назначение которого отличить префиксную функцию от постфиксной в момент объявления и определения функции. При вызове метода параметр передавать нет необходимости, так как при вызове префиксная функция отличается от постфиксной по расположению знака операции относительно операнда.

В случае перегрузки постфиксной функции в точку вызова возвращается старое значение.

Для реализации этого необходимо скопировать в теле функции объект, для которого вызван метод, в локальную переменную и вернуть его из функции, предварительно изменив значение текущего объекта, т.е. объекта для которого вызван метод. Т.к. объект передастся, в метод через указатель, его значение изменится и будет доступно при дальнейшей работе с ним.

Если постфиксная функция перегружена как внешняя функция, то объект необходимо передавать через указатель или по ссылке, для того что бы его значение изменилось.

Перегрузим в классе *A* постфиксный инкремент как метод класса и постфиксный декремент как дружественную функцию.

```
A operator ++(int) { A tmp = *this; a++; return tmp; }
A friend operator--(A& ob, int) { A tmp = ob; ob.a--;
return tmp; }
```

Вызовем написанные функции, и выведем на экран значение изменённого поля.

```
printf("ob_A1\t"); ob_A1.print();
printf("ob_A1++\t"); (ob_A1++).print();
printf("ob_A1\t"); ob_A1.print();
printf("ob_A1--\t"); (ob_A1--).print();
printf("ob_A1\t"); ob_A1.print();
```

Результат работы фрагмента кода, приведён на рисунке 5.

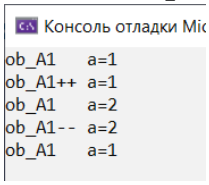



Рис. 5. Результат работы перегруженных постфиксных функций для класса *A*

Из результатов работы видно, в точку вызова вернулось старое значение, а уже в следующем операторе было использовано новое значение.

Таким образом, при перегрузке постфиксной и префиксной функций необходимо учесть, их отличие в возвращаемом значении.

Т.к. префиксная функция возвращает в точку вызова новое (измененное значение), то необходимо вернуть из функции объект, предварительно изменив его значение.

***Перегрузка операции присваивания***

Если в классе отсутствует явная перегрузка операции присваивания, то она будет сгенерирована автоматически.

Назначение операции - копирование полей объекта источника (правый операнд) в поля объекта приёмника (объект вызвавший метод, левый операнд).

В качестве возвращаемого значения указывают ссылку на объект, для которого функция вызвана.

Перегрузим операцию присваивания в описанном ранее классе *A*. Работа перегруженной операции, сгенерированной по умолчанию, была продемонстрирована в примере использования перегруженного оператора +.

Перегрузим явно оператор присваивания в классе *A* и продемонстрируем его работу.

```
A& operator=(const A& ob) { a = ob.a; return*this; }
    . . .
printf("ob_A1\t");        ob_A1.print();
printf("ob_A2\t");        ob_A2.print();
printf("ob_A1=ob_A2\n"); ob_A1= ob_A2;
printf("ob_A1\t");        ob_A1.print();
```

Результат работы приведенного фрагмента кода, представлен на рисунке 6.

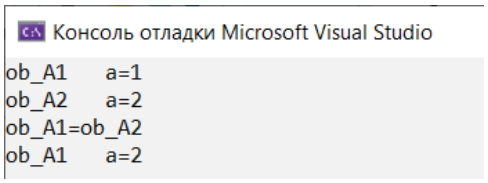



Рис. 6. Результат работы перегруженного оператора = в классе *A*

Как видно из рисунка 6, значения полей объекта `ob_A2` успешно скопированы в поля объекта `ob_A1`.

Присвоение объекта одного класса объекту другого класса возможно только после переопределения соответствующего оператора, по умолчанию такая функция сгенерирована не будет.

Перегрузим в классе *B* функцию `operator=` для реализации возможности присвоения объектам класса *B* объектов класса *A*.

```
B& operator =(const A& ob)
{ b = ob.a; return *this; }
```

Продемонстрируем результат работы перегруженного оператора присваивания.

```
//printf("-------ob_B=ob_A-----------\n");
ob_A1.print();
B b(9);
```

```
    b.print();
    b = ob_A1;/* без перегруженного оператора =, эта строчка приведёт к ошибке*/
    b.print();
```

Результат работы представленного фрагмента, представлен на рисунке 7.

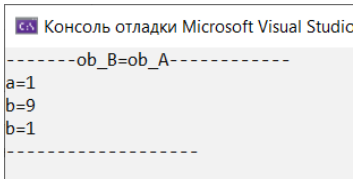



Рис. 7. Демонстрация работы перегруженного в классе *B* оператора =

## *Лабораторная работа № 1.*

## *Основы объектно-ориентированного программирования*

***Цель работы.***

Изучение основ объектно-ориентированного программирования, понятий класс, объект, абстракция, инкапсуляция, знакомство с модификаторами доступа, перегрузкой методов, дружественными функциями.

***Постановка задачи.***

Описать класс треугольник, содержащий четыре вещественных поля, три стороны и угол между двумя сторонами. Описать два конструктора: по умолчанию и с параметрами. Параметров должно быть три: две стороны и угол между ними, написать *Get* и *Set* методы для полей. Методы вычисления третей стороны, периметра и площади треугольника. Предусмотреть своевременное вычисление третей стороны. Перегрузить операторы умножения и сложения, префиксный и постфиксный инкремент и декремент. Создать массив объектов, динамический объект, набор объектов. Продемонстрировать работу всех описанных методов на созданных объектах. Один из перегруженных операторов реализовать в виде дружественной функции.

Класс треугольник описать в отдельном модуле, осуществить раздельную компиляцию.

В этой и всех последующих работах соблюдать правила именования идентификаторов.

***Контрольные вопросы***

1. Что такое объект?
2. Что такое класс?
3. Дня чего используют модификаторы доступа?
4. Какие вы знаете модификаторы доступа?
5. Напишите синтаксис создание объекта.
6. Что такое конструктор?
7. Какие виды конструкторов существуют?
8. Сколько конструкторов может быть описано в классе?
9. Что такое метод?
10. Что такое дружественные функции?
11. Что такое указатель *this*?
12. Может ли программист перенастроить указатель *this*?
13. Возможен ли доступ к указателю *this* из *main* функции?
14. Что такое деструктор?
15. Напишите синтаксис перегрузки оператора.
16. Сколько и почему именно столько параметров нужно передать в метод класса, если он является перегруженным бинарным оператором?
17. Сколько и почему именно столько параметров нужно передать в метод класса, если он является перегруженным унарным оператором?
18. Сколько и почему именно столько параметров нужно передать дружественной функции если она является перегруженным бинарным оператором?
19. Сколько и почему именно столько параметров нужно передать дружественной функции класса, если она является перегруженным унарным оператором?
20. Что общего и в чём отличия дружественной функции и метода класса?

***Задания для самостоятельной работы***

1) Опишите класс *Worker*, в котором будут следующие *private* поля - *name* (имя), *age* (возраст), *salary* (зарплата) и следующие *public* методы *setName, getName, setAge, getAge, setSalary, getSalary*. Создайте 2 объекта этого класса со значениями полей: «Иван», возраст 25, зарплата 10000 и «Василий», возраст 26, зарплата 20000. Выведите на

экран сумму зарплат Ивана и Василия. Выведите на экран сумму возрастов Ивана и Василия.

2) Дополните класс *Worker* из предыдущей задачи *private* методом *checkAge*, который будет проверять возраст на корректность (от 1 до 100 лет). Этот метод должен использовать метод *setAge* перед установкой нового возраста (если возраст не корректный - он не должен меняться).

3) Опишите класс *User*, в котором будут следующие *protected* поля: *name* (имя), *age* (возраст), *public* методы *setName, getName, setAge, getAge.*

4) Опишите класс *Worker*, который наследует от класса *User* и вносит дополнительное *private* поле *salary* (зарплата), а также *public* методы *getSalary и setSalary*.

5) Опишите класс *Student*, который наследует от класса *User* и вносит дополнительные *private* поля стипендия, курс, а также геттеры и сеттеры для них.

6) Опишите класс *Driver* (Водитель), который будет наследоваться от класса *Worker* из предыдущей задачи. Этот метод должен вносить следующие *private* поля: водительский стаж, категория вождения (*A, B, C*).

7) Построить систему классов для описания плоских геометрических фигур: круга, квадрата, прямоугольника. Необходимо предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и поворота на заданный угол.

8) Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса, создания и уничтожения объектов этого класса.

9) Составить описание класса для работы с цепными списками строк (строки произвольной длины) с операциями включения в список, удаления из списка элемента с заданным значением данного, удаления всего списка или конца списка, начиная с заданного элемента.

10) Составить описание класса для объектов-векторов, задаваемых координатами концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

11) Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на плоскости, изменения размеров, построения наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

12) Составить описание класса для определения одномерных массивов целых чисел (векторов). Предусмотреть возможность обращения к отдельному элементу массива с контролем выхода за пределы индексов, возможность задания произвольных границ индексов при создании объекта и выполнения операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов, умножения и деления всех элементов массива на скаляр, печати (вывода на экран) элементов массива по индексам и всего массива.

13) Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы индексов, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, печать (вывод на экран) элементов массива и всего массива.

14) Составить описание класса многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Предусмотреть методы для вычисления значения многочлена для заданного аргумента, операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена, печать (вывод на экран) описания многочлена.

## Наследование

### *Одиночное наследование*

### *Теоретические сведения и методические указания к работе*

Механизм наследования позволяет строить иерархии классов. При этом дочерние (производные классы) обладают атрибутами (полями) и поведением (методами) родительских (базовых) классов. Наследуются поля и методы, объявленные с модификаторами доступа *public* и *protected*. В классе наследнике могу быть объявлены дополнительные поля и методы, методы базового класса в классе наследнике могут быть перегружены.

При объявлении класса наследника, после его имени через двоеточие указывается ключ доступа, далее имя базового класса.

```
class <имя класса> : <ключ доступа> <имя базового класса> {описание класса наследника};
```

Например, так объявлен класс *C*, наследник класса *A*.

```
class C :public A
{ <описание класса C>}
```

Класс наследник может иметь несколько базовых классов, в этом случае, они перечисляются через запятую, каждый со своим ключом.

```
class C :public A, public B
{<описание класса C>};
```

Наследование называется *простым*, если производный класс имеет один базовый класс. Если базовых классовых несколько, наследование называется *множественным*.

По умолчанию используют ключ доступа *private*.

Ключи доступа и их влияние на члены класса наследника унаследованные из родительского класса, объявленные с различными модификаторами доступа, представлены в таблице1.

Таблица 1

***Ключи доступа***

| Ключ доступа | Спецификатор в базовом классе | Доступ в производном классе |
|---|---|---|
| `private` | `private` | нет |
| | `protected` | `private` |
| | `public` | `private` |
| `protected` | `private` | нет |
| | `protected` | `protected` |
| | `public` | `protected` |
| `public` | `private` | нет |
| | `protected` | `protected` |
| | `public` | `public` |

Элементы, объявленные с модификатором `private` в родительском классе не доступны в базовом при любом ключе.

При необходимости унаследовать элементы и в тоже время сделать их закрытыми для внешнего использования их объявляют в родительском классе с модификатором `protected`.

Элементы, которые были объявлены в базовом классе со спецификатором `public` в классе наследнике получают спецификатор доступа, соответствующий ключу доступа.

***Правила наследования некоторых методов класса***

Конструкторы и перегруженная операция присваивания не наследуются. Поэтому в классе наследнике должны быть описаны собственные конструкторы.

Если в конструкторе производного класса нет явного вызова конструктора базового класса, то будет сгенерирован вызов конструктора базового класса по умолчанию.

Если имеется несколько уровней наследования, то вызов конструкторов начинается с самого верхнего уровня.

Если класс наследуется от нескольких классов, то их конструкторы вызываются в порядке объявления.

```
class C :public A, public B
```

```
{ private: int c;
public:
C(int a1, int b1, int c1) :A(a1), B(b1) { c = c1;};
```

При необходимости вызвать в конструкторе класса наследника конструктор с параметрами базового класса, нужно это сделать явным образом с передачей всех необходимых параметров.

Деструкторы не наследуются.

Если в производном классе не описан деструктор, он генерируется автоматически и при его вызове происходит вызов деструкторов всех базовых классов.

Если деструктор описан в классе наследнике, вызывать в нём явно деструкторы базовых классов не нужно, это произойдёт автоматически.

Вызов деструкторов происходит в порядке обратном вызову конструкторов. Т.е. сначала происходит вызов деструктора соответствующего класса, а потом деструкторов родительских классов.

*Конструкторы выполняются в порядке наследования, а деструкторы* – в обратном порядке.

При написании методов класса наследника, выполняющих действия базового класса, предпочтительнее не дублировать код, а вызывать методы базового класса. Кроме сокращения кода это ведёт к упрощению модификации кода, при необходимости внести изменения их нужно будет внести в одном методе.

В классе наследнике может быть выполнена перегрузка методов родительского класса, при необходимости расширить и/или изменить их функционал. Доступ к переопределённому методу базового класса для объекта производного класса выполняется через уточнение имени.

### *Множественное наследование*

Наследование называется множественным, если класс наследник имеет несколько родительских классов. При этом может возникнуть конфликт имён, если в базовых классах есть одноимённые элементы, конфликт устраняется с помощью операции доступа к области видимости.

Рассмотрим пример множественного наследования.

```
class A {
 protected: int a;
```

```
 public:
A() { a = 2; };
A(int a1) { a = a1; };
void print() { printf("a=%i \n", a); };
};// class A
class B
{protected: int b; public: B() { b = 3; };
B(int b1) { b = b1; };
void print() { printf("a=%i \n", b); }
}; // class B
class C :public A, public B
// класс C – наследник классов A и B
{private: int c;
public:
C(int a1, int b1, int c1) :A(a1), B(b1) { c = c1; };
C() { c = 4; };
void print() { printf("a=%i b=%i c=%i\n", a, b, c); };
}// class C;
int main()
{C ob1(1,2,3); ob1.print();
ob1.A::print();
/* вызов метода print родительского класса A для объекта
класса наследника C*/
…}
```

В строчке `ob1.A::print();` вызван метод родительского класса для объекта класса наследника.

Результат работы, приведённого выше фрагмента кода Представлен на рисунке 8.

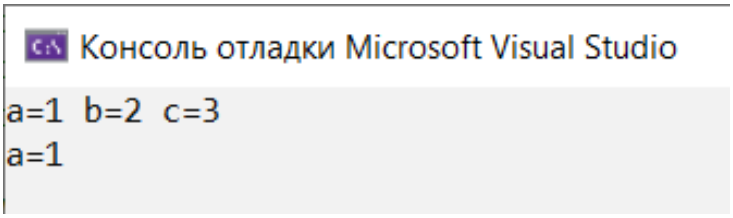



Рис. 8. Вызов метода базового класса для объекта класса наследника

Если у базовых классов есть общий предок, то его поля будут наследованы дважды, чтобы этого избежать общий предок объявляют виртуальным.

### *Лабораторная работа № 2. Наследование*

***Цель работы.***

Совершенствование навыков объектно-ориентированного программирования. Изучение механизма наследования в языке *C++*, переопределение методов. Совершенствование навыка написания дружественных функций.

***Постановка задачи.***

Разработать класс пирамида, наследник класса треугольник, описанного в предыдущей лабораторной работе. Класс пирамида должен содержать одно дополнительное поле – высоту пирамиды. Описать метод возвращающий объём пирамиды.

Создать функцию дружественную классу Пирамида. Функция должна вычислять и возвращать разность или сумму (выбор по желанию разработчика) объёмов двух пирамид. Перегрузить для этого соответствующий оператор. Продемонстрировать работу функции на объектах соответствующего класса.

Перегрузить оператор присваивания. Продемонстрировать работу метода и результат работы. Для этого вывести на экран значения полей объекта источника и объекта приёмника до и после копирования.

Создать массив объектов класса Пирамида.

Для объектов массива вызвать методы класса. Результаты работы методов вывести на экран.

***Контрольные вопросы***

1. В каких случаях прибегают к механизму наследования?
2. В каком случае наследование называют одиночным?
3. В каком случае наследование называют множественным?
4. Происходит ли наследование конструкторов?
5. Какому правилу подчиняется вызов конструкторов при создании объекта класса потомка?
6. Какому правилу подчиняется вызов деструкторов при удалении из памяти объекта класса потомка?
7. Что такое ключ наследования?

8. Какой модификатор доступа будет у члена класса наследника, если в родительском классе он был объявлен с модификатором *private*, а наследование произведено с ключом *public*?

9. Какой модификатор доступа будет у члена класса наследника, если в родительском классе он был объявлен с модификатором *protected*, а наследование произведено с ключом *public*?

10. Какой модификатор доступа будет у члена класса наследника, если в родительском классе он был объявлен с модификатором *public*, а наследование произведено с ключом *private?*

***Задания для самостоятельной работы***

Описать базовый класс и класс наследник. В *main* функции должна быть возможность создания и удаления объектов каждого из классов, вывода на экран значений полей.

Описать методы: `Move()` - перемещения объекта на плоскости, `Compare(T&,T&)` - сравнения площадей объектов, `IsIntersect(T&,T&)` - определения факта пересечения объектов, `IsInclude(T&,T&)` - определения факта включения одного объекта в другой, при необходимости можно разработать дополнительные методы.

Далее в задании по вариантам, перечислены необходимые к реализации классы, родительский и наследник, соответственно.

1) Треугольник, квадрат.
2) Квадрат, прямоугольник.
3) Треугольник, прямоугольник,
4) Прямоугольник, пятиугольник.
5) Треугольник, четырёхугольник.
6) Квадрат, четырёхугольник.

В следующих вариантах необходимо определить методы: отображения *ShowBin()* – отображения двоичного представления числа, *ShowOct()* - отображения восьмиричного представления числа, *ShowDec()* - отображения десятичного представления числа, *ShowHex()* - отображения шестнадцатиричного представления числа, operator+(*T&,T&*)

7) Строка символов, Двоичная строка.
8) Строка символов, Восьмеричная строка.
9) Строка символов, Десятичная строка.
10) Строка символов, Шестнадцатеричная строка.

## Полиморфизм

Работу с объектами классов обычно ведут через указатели.

При открытом наследовании указатель на базовый класс можно настроить на любой производный.

При этом вызов методов для объекта происходит в соответствии с типом указателя, а не типом объекта, на который он настроен. Т.к. ссылки на методы происходят во время компоновки программы. Этот процесс называется *ранним связыванием.*

Объявим указатели на объекты базового и производного классов и настроим их на объекты соответствующих классов.

```
A ob_A(7);
B ob_B(6);
C ob;
A* pA = &ob_A;
B* pB = &ob_B;
C* pC=& ob;
```

Далее вызовем через объявленные указатели метод *print();*

Для вызова метода через указатель, настроенный на объект, этот указатель необходимо разыменовать, далее вызвать метод обычным способом через точку. Эти два действия можно заменить одной операцией ->.

*<указатель> -> <имя метода>*

Следующий пример иллюстрирует оба подхода.

```
pA->print();
pB->print();
(*pC).print();
```

Результаты работы фрагмента кода приведены на рисунке 9.

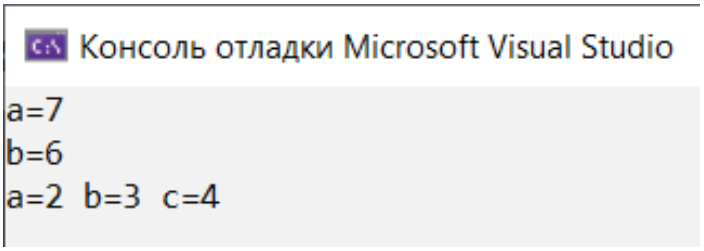



Рис. 9. Результат вызова метода через указатель на объект

Перенастроим указатель на родительский класс на объект класса наследника и снова вызовем метод *print();*

```
pA = &ob;
pA->print();
```

Результат работы эго фрагмента кода приведён на рисунке 10.

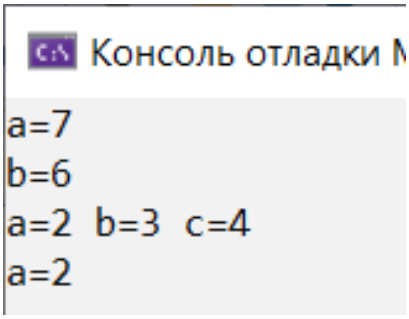



Рис. 10. Вызов метода родительского класса для объекта класса наследника

В приведенном выше фрагменте программы для объекта класса наследника был вызван метод родительского класса, так как вызов метода производится по типу указателя, по причинам, описанным выше.

Настройка указателя на класс наследник на объекты базового класса запрещена. В противном случае открылась бы возможность вызывать методы наследника для объектов родительского класса. А это может привести к ошибке, т.к. в методах класса наследника возможно обращение к полям отсутствующим в родительском классе.

Что бы вызвать через указатель метод класса указуемого объекта, можно воспользоваться приведением типов, но это не всегда возможно, т.к. в разные моменты выполнения программы указатель может быть настроен на объекты разных классов.

Что бы реализовать вызов метода указуемого через указатель, необходимо воспользоваться механизмом *позднего связывания*. При этом ссылки на метод происходят в момент вызова метода, при выполнении программы, в зависимости от типа конкретного объекта.

Механизм позднего связывания реализуется на виртуальных методах. Что бы метод был виртуальным его необходимо объявить с ключевым словом *virtual*.

*virtual void metod(int a);*

Если в базовом классе метод определить, как виртуальный, то метод класса наследника с тем же именем и набором параметров автоматически становится виртуальным, а с отличным набором параметров обычным, не виртуальным.

Виртуальный метод наследуется. Т.е. без необходимости его переопределять не нужно.

Если виртуальный метод переопределён в классе наследнике, то его объекты могут вызвать метод базового класса через операцию доступа к области видимости.

*Механизм позднего связывания* реализован следующим образом. Для каждого класса, содержащего хотя бы один виртуальный метод, на шаге компиляции создаётся таблица виртуальных методов, в которой хранятся адреса виртуальных методов в порядке их объявления в классе.

Каждый объект класса, содержащий виртуальные методы, имеет дополнительное поле ссылки на таблицу виртуальных методов (*vptr*), это поле получает значение во время работы конструктора при создании объекта.

Во время компиляции ссылка на виртуальные методы заменяется на обращение к таблице виртуальных методов (*vtbl*) через *vptr* объекта, во время выполнения программы, адрес метода выбирается из таблицы.

Процесс вызова виртуального метода происходит медленнее чем обычного из-за дополнительного этапа получения адреса. Поэтому нет смысла объявлять виртуальными методы, остающиеся неизменными на всей иерархии класса и те методы, которые не будут использованы в производных классах.

Виртуальными рекомендуется делать деструкторы, это гарантирует корректное освобождение памяти.

В тоже время использование виртуальных методов добавляет гибкости программе.

Механизм использования виртуальных методов проявляется только если мы работаем с объектами через указатели или ссылки.

Такой объект называют полиморфным. Проявление *полиморфизма* заключается в вызове различных методов, а, следовательно, выполнении различных действии при выполнении одинаковых строчек

кода. Т.к. при позднем связывании метод вызывается по типу указуемого, а не указателя.

Проиллюстрируем выше сказанное на примере.

Для этого внесём в написанный ранее код одно изменение, объявим метод *print* в родительском классе *A* виртуальным.

```
class A
{protected:
int a;
public: A() { a = 2; };
A(int a1) { a = a1; };
virtual void print() { printf("a=%i \n", a); };
};
```

Снова вызовем метод *print* для объекта класса наследника *ob*, через указатель на родительский класс *pA*.

```
A ob_A(7);
B ob_B(6);
A* pA = &ob_A; B* pB = &ob_B; C* pC=& ob;
(*pC).print(); pA = &ob; pA->print();
```

Результат работы программы представлены на рисунке 11. Этот результат отличается от предыдущего, несмотря на то что отличий в вызове методов, объявлении и настройке указателей нет.

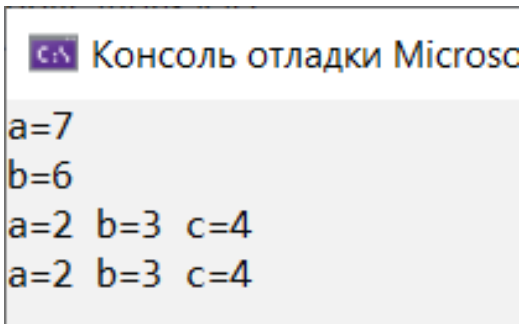



Рис. 11. Результат вызова полиморфного метода через указатель на родительский класс

Разница в результате работы объясняется, тем, что в последнем примере метод *print* в родительском классе объявлен виртуальным. Поэтому через указатель на родительский класс, настроенный на объект производного класса был вызван метод класса наследника, т.е. вызов произошёл по типу класса указуемого, а не указателя.

Виртуальный метод должен быть определён. Если не предполагается его реализация в этом классе, а только в классе потомке, то такой метод должен содержать признак нуля вместо тела. Такой метод называется *чисто виртуальным*. Приведём пример класса содержащего такой метод.

```
class Abs
{
virtual void print()=0;
};//Abs
```

Класс, содержащий хотя бы один чисто виртуальный метод, называется *абстрактным.* Такие классы предназначены для задания общих представления о классе. Абстрактные классы используют в качестве базовых, предполагая реализацию методов в классах наследниках. Нельзя создать объект абстрактного класса. Вызов чисто виртуального метода приведет к ошибке.

```
Abs N;
```

Объект *N* класса *Abs* не будет создан. Попытка его создания приведёт к ошибке. Т.к. по Правилам языка не допускается использование объекта абстрактного класса, каковым является класс *Abs,* содержащий чисто виртуальный метод `Abs::print()`.

## *Лабораторная работа № 3. Полиморфизм*

***Цель работы.***

Совершенствование навыков объектно-ориентированного программирования, использования механизма наследования.

Изучение свойства полиморфизма, механизмов раннего и позднего связывания.

***Постановка задачи.***

Дополнить класс треугольник из предыдущей лабораторной работы виртуальным методом *Show()*. Наделив его следующим функционалом, отображение на экране полной информации об объекте, значение сторон треугольника и известного угла, периметра и площади. В классе пирамида метод *Show()* должен быть переопределён, с расширением функционала, кроме перечисленного, необходимо вывести на экран значение высоты и объёма пирамиды.

Объявить указатель на объект класса треугольник и указатель на объект класса пирамида. Настроить объявленные указатели на объекты соответствующих классов.

Вызвать метод *Show()* через объявленные указатели.

Перенастроить указатели на объекты не соответствующих классов. Закомментировать строчку дающую ошибку, объяснить эту ошибку в комментариях.

Снова вызвать метод *Show()*. Объяснить полученный результат.

***Контрольные вопросы***

1. Какую функцию называют виртуальной?
2. Поясните механизм раннего и позднего связывания.
3. Что такое таблица виртуальных методов?
4. Каким образом осуществляется связь с таблицей виртуальных методов?
5. Каким образом определяется адрес нужного метода в таблице виртуальных методов?
6. В чём преимущества позднего связывания?
7. В чём недостатки позднего связывания?
8. В каком случае виртуальный метод родительского класса останется виртуальным в классе наследнике?
9. Какие методы нет смысла делать виртуальными?
10. При вызове виртуального метода через указатель, на объект, метод будет вызван по типу указателя или указуемого?

***Задания для самостоятельной работы***

1) Описать иерархию классов: учащийся, ученик, студент, работающий студент, метод вывода информации об объекте сделать полиморфным.

2) На основе класса *Shape* (фигура) описать класс *Ellipsoid* (эллипсоид). Эллипсоид будет характеризоваться данными *x, y, z* (наследуются от *Shape*) и *a,b,c* – полуоси. В классе *Ellipsoid* переопределить функцию для расчета объема. Объем эллипсоида вычисляется по формуле $V = 4\pi abc/3$.

3) Создать абстрактный базовый класс с виртуальной функцией Площадь. Создать произвольные классы Прямоугольник, Круг, Прямоугольны треугольник, Трапеция со своими функциями площади.

Для проверки определить массив ссылок на абстрактный класс, которым присваивается адреса различных объектов.

4) Разработать иерархию классов. В иерархии должно быть не менее двух уровней и не менее четырех классов (вместе с базовым классом). Каждый класс-потомок должен иметь хотя бы один собственный атрибут, и как минимум один собственный метод. Предусмотреть не менее одного виртуального метода, например, метод вывода информации об объекте на экран. В каждом классе-потомке должен быть описан собственный конструктор. В основной программе надо привести небольшой демонстрационный пример, показывающий возможности работы с классами-потомками: создание экземпляра класса-потомка, доступ к атрибутам и методов класса-предка, порядок вызова конструкторов и т.д.

5) Создать абстрактный базовый класс объектов на декартовой плоскости *CShap*e (фигура). Разработать и реализовать иерархию классов конкретных фигур: *CPoint, CCircle* и т.д., не менее четырёх классов наследников. При этом: во всех классах должен быть реализован метод *Square(),* вычисляющий и возвращающий площадь объект и метод *Show(),* который отображает на экране информацию в текстовом виде о данном экземпляре (не менее трех характеристик; у точки есть имя и координаты, у окружности ещё есть радиус и площадь, у ломаной - длина и так далее.

6) Создать базовый класс *Array* с виртуальными методами сложения и поэлементной обработки массива. Разработать производные классы *AndArra*y и *OnArray*. В первом классе операция сложения реализуется как пересечение множеств, а поэлементная обработка представляет собой извлечение квадратного корня. Во втором классе операция сложения реализуется как объединение, а поэлементная обработка-вычисление логарифма.

7) Создать иерархию классов Шахматная фигура – абстрактный класс, содержащий поле – цвет. Создать производные классы все фигуры, содержащие свое название и координаты позиции на доске. Определить конструктор копирования, оператор присваивания через соответствующие функции базового класса. Продемонстрировать работу классов.

8) Создать класс *3D* фигура, и производные классы шар, конус, цилиндр и куб. В классе должны присутствовать виртуальные методы вычисления объёма и вывода информации об объекте на экран. Объявить массив указателей на базовый класс, настроить элементы массива на разнотипные объекты. Используя эти указатели продемонстрировать работу упомянутых выше методов.

9) Создать абстрактный класс - млекопитающие. Определить производные классы - животные и люди. У животных определить производные классы собак и коров. Определить виртуальные функции описания человека, собаки и коровы.

10) Создать абстрактный класс *Norm* с виртуальной функцией вычисления нормы и модуля. Определить производные классы *Complex, Vector3D* с собственными функциями вычисления нормы и модуля. (Модуль для комплексного числа вычисляется как корень из суммы квадратов действительной и мнимой частей; норма для комплексных чисел вычисляется как модуль в квадрате. Модуль вектора вычисляется как корень квадратный из суммы квадратов координат; норма вектора вычисляется как максимальное из абсолютных значений координат).

## Шаблоны классов

Шаблоны классов дают возможность определить общую структуру класса, создавая параметризированные классы. Такие классы можно применять к любому типу данных, передаваемому в качестве параметра.

Этот подход используют при разработке контейнерных классов. *Контейнерные классы* предназначены для хранения определённым образом организованных данных с возможностью доступа к ним.

Синтаксис описания шаблона класса.

*template <описание параметров шаблона> определение класса;*

Перечисление параметров класса происходит через запятую. В качестве параметров класса могут выступать базовые или пользовательские типы данных, шаблоны и переменные.

Внутри класса шаблона параметр типа может применяться в любом месте где допустимо указание типа.

Например, так для класса шаблона.

```
template <class Data> class Konteyner
{
int count;// количество элементов
Data* head;// указатель на вершину
int index;// номер текущего элемента}
}
```

Область действия параметра шаблона - от точки описания до конца шаблона.

Для создания конкретного объекта, конкретного класса (описанного при помощи шаблона) необходимо выполнить следующие действия. После имени шаблона класса, перед именем объекта в угловых скобках перечислить аргументы класса, в соответствии с параметрами шаблона.

*имя шаблона <аргументы> имя объекта [(параметры конструктора)];*

Например, так

```
T_Stack <int>T_S1(5);
```

Будет создан объект *T_S1* класса *T_Stack*, в объект может быть помещено пять данных типа *int.*

Имя шаблона вместе с аргументами можно рассматривать как уточнённое имя класса.

Пример обращения к методам класса, объекты, которого помещены в стек.

Здесь *T_S_Kv* – объект класса контейнера, в который помещены объекты класса квадрат.

```
(T_S_Kv.POP()).Sq());
```

или

```
Kvadrat K = T_S_Kv.POP();
```

// извлекаем объект их стека

```
printf("%.3f % .3f\n", K.Get_a(), K.Sq());
```

В последней строке кода приведен пример использования извлеченного значения, сохраненного в переменную *K*, в качестве параметра функции.

Шаблоны классов представляют эффективное средство обращения с данными.

Однако, программа, использующая шаблоны, содержит полный код для всех порожденных типов, что увеличивает размер исполняемого файла.

## Контейнерные классы

Контейнерные классы предназначены для хранения информации, с организованной определённым образом доступом к элементам. Стандартная библиотека классов *C++* содержит классы, позволяющие реализовать основные структуры данных массивы – векторы *vector*, очереди *queue*, разновидности списков *list*, очереди с приоритетом (*priority_ queue*) и другие.

При работе выбор класса контейнера определяется условиями задачи, и соответственно операциями. Которые планируется осуществлять с объектами класса.

На объектах класса *Vecto*r эффективно реализован произвольный доступ к элементам, вставка и удаление элемента в конец вектора.

Двусторонняя очередь позволяет реализовать произвольный доступ к элементам, вставку и удаление элементов с двух концов.

Список позволяет вставлять и удалять элементы из произвольной части, но запрещает произвольный доступ к элементам.

Ниже приведены примеры работы с объектами упомянутых выше классов.

Для создания объектов и работы с ними необходимо подключить соответствующие заголовочные файлы.

```
#include <vector>
#include <list>
#include <deque>
int main()
{
    vector <int> v;
    for (int i = 0; i < 5; i++) v.push_back(i);
    cout << v[3] << endl; ;
```

В результате работы приведённого фрагмента кода на экране будет отображено число 3 – элемент вектора *v* с индексом 3.

Создадим объект класса `list.`

```
list <int> L;
```

Заполним его с помощью метода *push_back().*

```
for (int i = 0; i < 5; i++) L.push_back(i);
```

Настроим итератор *I* на начало списка, итератор можно рассматривать как указатель на элемент контейнера.

```
list<int>::iterator I = L.begin();
```

Вставим в список элемент со значением -10, предварительно, сдвинув итератор.

```
L.insert(++I,-10);
```

Установим итератор на начало списка и выведем значения на экран.

```
I = L.begin();
while (I != L.end())
{
cout << *I++; cout << " ";
}
```

Результат будет таким: 0 -10 1 2 3 4

Упорядочим элементы списка, использовав метод *sort*. И снова выведем на экран.

```
L.sort();
I = L.begin();
while (I != L.end())
{cout << *I++; cout << " ";
}
```

На экране будет следующая последовательность: -10 0 1 2 3 4

Создадим объект касса `deque`.

```
deque <int>q;
```

Запомним контейнер квадратами номеров элементов в порядке поступления начиная с 1.

```
for (int i = 1; i < 5; i++) q.push_back(i*i);
```

Извлечём элементы с конца списка, предварительно отобразив их на экране.

```
while (!q.empty()) { cout << q.back() << " "; q.pop_back(); }
```

На экране будет отображена следующая последовательность: *16 9 4 1 0*.

Как уже было сказано, контейнерные классы предназначены для хранения определённым образом организованных данных с возможностью доступа к ним.

При создании пользовательского класса для хранилища данных необходимо предусмотреть два момента: создание самого объекта, выделение места под блок хранимых данных.

Если предполагается, что поля объекта и хранилище данных находятся в разных местах, то класс обязательно должен содержать поле указатель на вершину хранилища, захват памяти под хранилище и настройка указателя производятся в конструкторе.

Пример такого объявления.

```
class Stack
{
int count;// количество элементов
int* head;// указатель на вершину
int index;// номер текущего элемента
public:
Stack(int C)
{ count = C; head = new int[C]; index = 0; };
```

При добавлении элементов в хранилище и извлечении элементов из хранилища необходимо осуществлять проверку хранилища на пустоту.

Рекомендуется для этих целей создать методы, возвращающие логическое значение. Например, значение *true*, если стек полон. Для того чтобы убедится в том полон стек или пуст, достаточно проверить значение поля, в котором хранится номер текущего элемента.

Если индекс равен количеству элементов в стеке, то стек полон, если индекс равен 0, стек пуст, т.к. в текущий момент времени значение индекса равно номеру следующего элемента.

```
bool StackIsFull()
{
if (index==count) return true; else return false;
}
bool StackIsEmpty()
{
if (index <1) return true; else return false;
}
```

Контейнерный класс должен содержать методы добавления и извлечения элементов из него. Если хранилище организовано по принципу стека, то эти методы должны быть оформлены следующем образом.

```
void Push(int a)
{head[index] = a; index++; };
int POP()
{return head[--index];}
```

Вызову методов добавления и извлечения элементов из хранилища, должна предшествовать проверка хранилища на пустоту или полное заполнение.

Создадим объект контейнерного класса *Stack*, фрагменты описания которого были представлены выше.

```
Stack St(5);
```

Заполним хранилище данными и извлечем их.

```
for (int i = 0; i < 5; i++) {if (!St.StackIsFull())
St.Push(i); }
while (!St.StackIsEmpty()) { printf("%i", St.POP()); }
```

Результат работы представленного фрагмента кода представлен на рисунке 12.

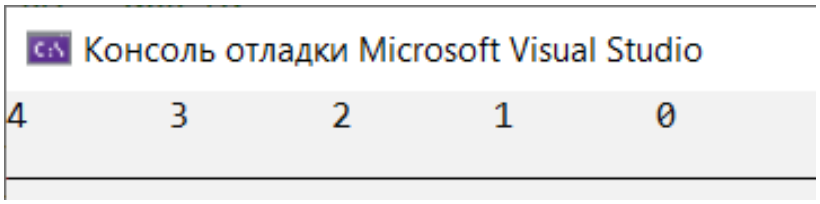



Рис. 12. Извлечение элементов из стека

Из приведенных на рисунке результатов работы кода видно, что данные извлечены по правилу стека: первый вошел – последний вышел.

Попробуем положить в хранилище большее количество элементов. Для этого изменим количество итераций цикла *for*.

```
Stack St(5);
for (int i = 0; i < 7; i++) { if (!St.StackIsFull())
St.Push(i); else printf("StackIsFull\n"); }
```

```
while (!St.StackIsEmpty()) { printf("%i    ",
St.POP()); }
```

Результат представлен на рисунке 13.

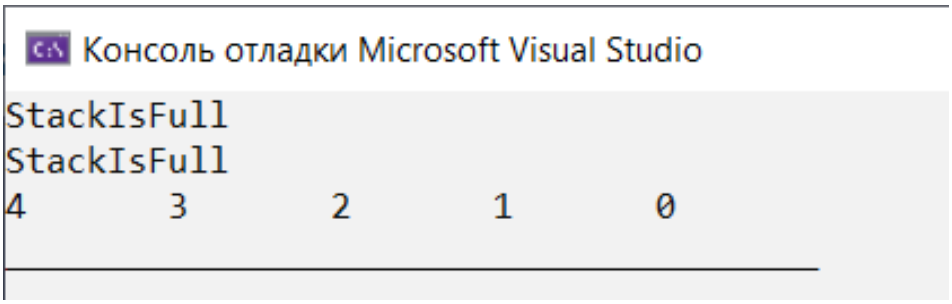



Рис. 13. Результат попытки переполнения стека

Из результатов работы видно, что количество и порядок элементов в стеке не изменились. При попытке записи в полностью заполненное хранилище, на экране было отображено соответствующее сообщение.

Если планируется совместное хранение объекта (т.е. его полей) и хранилища данных, то в классе необходимо осуществить перегрузку операции `new`, как метод класса. Так как при создании динамического объекта сначала вызывается оператор `new`, выделяющий динамическую память, а затем вызывается конструктор.

Если `new` перегружается в классе, то он считается `static` методом, не зависимо от того указан явно модификатор `static` или нет.

Рассмотри перегрузку оператора `new` подробнее.

```
void* operator new (size_t sizeOb, int count);
```

Оператор `new` возвращает нетипизированный указатель `void*` и получает на вход два параметра, первый – размер объекта типа `size_t`, возвращаемый функцией `sizeof`, второй – количество элементов в хранилище.

Ниже приведён пример описания перегруженной функции *new*.

```
void* Stack:: operator new (size_t sizeOb, int
count)
{return new char[sizeOb +sizeof(Type)*count];}
```

## *Лабораторная работа № 4. Классы контейнеры*

***Цель работы.***

Совершенствование навыков объектно-ориентированного программирования. Изучение способов создания контейнерных классов.

Изучения возможностей описания и использования шаблонов классов.

***Постановка задачи.***

Описать шаблон класса `Stack`, позволяющий хранить данные по принципу стека.

Класс должен содержать поле – указатель на вершину стека, а сам стек должен быть организован через динамический массив.

Создать экземпляры этого класса для следующих типов данных: целочисленный, вещественный, квадрат или круг (класс, описанный в качестве примера в первой лабораторной работе).

Созданные таким образом стеки необходимо заполнить и вывести значения из них на экран. Для одного из объектов стека заполненного объектами класса квадрат (или круг) вызвать метод вычисления площади, результат работы метода отобразить на экране.

***Контрольные вопросы***

1. Что такое шаблон класса?
2. Для чего используют шаблоны классов?
3. Какие классы называют контейнерами?
4. Какие есть два подхода к организации контейнерных классов?
5. Каким образом создать объект класса с необходимым типом данных, описанного как шаблон?

Каким образом можно определить размер памяти, который необходимо выделить под объект, если управляющая часть, совмещена с хранилищем?

6. Если контейнерный класс, организован по принципу стека через массив, каким образом осуществляется доступ к элементам?
7. Каким образом осуществляется проверка контейнера на пустоту?
8. Каким образом осуществляется проверка контейнера на полное заполнение?
9. Как получить доступ к первому элементу хранилища, если хранилище совмещено с управляющей частью объекта?

***Задания для самостоятельной работы***

Разработать шаблон класса *A*. Далее приведены варианты заданий, во всех вариантах описать методы добавления и извлечения элементов из хранилища, обработку попытки добавления элемента в заполненное хранилище и попытки извлечения из пустого хранилища.

1) Одномерный динамический массив.
2) Двунаправленный список.
3) Двунаправленный список кольцевой.
4) Одномерный динамический массив.
5) Стек.
6) Бинарное дерево.
7) Очередь.
8) Двусторонняя очередь (вставка и удаление возможны с обеих концов очереди).
9) Множество (хранит элементы в отсортированном порядке, дублирование не допускается).
10) Разреженный массив.

# РАБОТА В ВИЗУАЛЬНОГЙ СРЕДЕ ПРОЕКТИРОВАНИЯ

## Объектно-ориентированное программирование на языке C#

*C#* это полностью объектно-ориентированный язык, поэтому полноценное изучение дисциплины объектно-ориентированное программирование без изучения этого языка невозможно.

Краткая справочная информация по языку программирования C#, приведена в приложениях практикума.

Вспомним основы объектно-ориентированного подхода, уделяя внимание синтаксису и приводя примеры на языке C#.

Инкапсуляция — это объединение в одной сущности данных и кода, обрабатывающего эти данные, при этом исключается вмешательство извне, и неправильное использование данных. Инкапсуляция представляет возможность скрывать детали реализации от пользователя, обеспечивая защиту данных.

Данные-состояния объекта должны быть специфицированы с использованием ключевого слова `private` (или `protected`, если предполагается наследование).

Основной единицей инкапсуляции в C# является класс, который определяет форму объекта. В классе описаны данные, а также код, который будет ими оперировать (методы). Код и данные, составляющие вместе класс, называют членами. Данные, класса, называют полями, или переменными экземпляра. Код, оперирующий данными, содержится в функциях-членах, методах класса.

Класс можно рассматривать, как пользовательский тип данных, а объекты, как переменные этого типа.

Описав класс, наделив его необходимыми атрибутами и поведением, можно создать необходимое число экземпляров этого класса – объектов.

В качестве примера объекта из обыденной жизни, который может быть описан на языке программирования с помощью класса, рассмотрим шариковую ручку. Она обладает следующими свойствами: длина, диаметр шарика, цвет пасты и т.д. за основной метод можно принять оставление следа на бумаге, т.е. то для чего ручка и предназначена.

При этом длина и диаметр - это свойства, задаваемые числом, а цвет можно задать строкой.

Описание класса на языке программирования *C#* начинается с ключевого слова *class,* далее следует имя класса, не объекта, а именно класса, описания по которому создаются экземпляры класса - объекты.

Описание класса заключено в фигурные скобки. Желательно что бы описание класса происходило внутри какого-либо пространства имен.

```
class PEN
{…
}
```

Переменная типа описанного класса создается по тем же правилам, что и переменные всех остальных типов.

```
<тип> <имя переменной>;
PEN myPen1;
```

Теперь переменную необходимо проинициализировать, по тем же правилам как происходит инициализация массивов в языке C#.

```
myPen1=new PEN ();
```

Объявление и инициализацию можно совместить.

```
PEN myPen1=new PEN();
PEN myPen2=new PEN();
```

Для организации доступа к свойствам, методам, и самим классам в *C#* используют следующие модификаторы доступа:

`public` – член объекта доступен везде, где есть доступ к самому объекту;

`private` – только внутри класса, за его пределами доступа нет, даже для потомков;

`protected` – только объекту и его потомкам.

Модификатор доступа может быть установлен не только для членов класса, но и для самих классов.

***Свойства***

Под свойствами в языке C# понимают не просто переменные, объявленные внутри класса, как в других языках, например, *C++*.

Данные класса всегда должны быть закрытыми. Поэтому они должны быть объявлены с модификаторами *private* или *protected.*

Свойства же должны быть открытыми, поэтому их описание начинается со слова *public* далее указывают тип данных и имя свойства.

Поля принято именовать с маленькой буквы, а свойства с большой.

```
private int dlina;
public int Dlina
{ get {return dlina;}
   set {dlina=value;}
}
```

Далее в фигурных скобках идет реализация акессоров, позволяющих организовать доступ к значению свойства на чтение (*get*) и запись (*set*).

Для реализации *get* необходимо использовать оператор *return,* указав после него значение, которое должно возвращать свойство.

В нашем случае это значение переменной *dlina*.

Аксессор *set* позволяет сохранять значение свойства, которое хочет установить пользователь.

Переменную `value` объявлять не нужно, это виртуальная переменная, которая всегда существует внутри фигурных скобок после ключевого слова *set* и имеет такой же тип, как и свойство.

При необходимости можно создать свойство только для чтения.

```
private int Dlina
{get {return dlina}
}
```

Или реализовать проверку вводимого значения, например.

```
private int Dlina
{
get {return dlina}
set { if (value>0 || value <100) dlina=value;}
}
```

Таким образом, обеспечивается защита данных.

Асессоры могут также иметь модификаторы доступа, по умолчанию они `public`.

Если аксессор объявляется с модификатором `private`, то мы получаем свойство для чтения, изменить значение, которого можно только внутри класса.

При обращении к свойству из другого класса необходимо писать полное имя, начав с имени объекта.

<имя объекта>.<имя свойства>

Внутри класса имя объекта писать не надо.

Сокращённый вариант объявления свойства.

```
public int  Dlina {get; set;}
```

***Методы***

Метод - это функция-член класса предназначенная для работы с данными, при этом метод может возвращать значение, которое может быть использовано за пределами класса или иметь тип возвращаемого значения `void`. Круглые скобки после имени метода обязательны, они могут быть пустыми, если метод не получает входных параметров. Наличие круглых скобок после имени, можно рассматривать как признак того что данный идентификатор – имя метода.

Перед именем метода указывают модификатор доступа и тип возвращаемого значения. Если метод ничего возвращать не должен, пишут `void`. Можно провести аналогию такой функции (функции, которая не возвращает значения) с процедурой в *Delphi*.

Далее в фигурных скобках следует реализация метода. Т.е. код, который будет выполнен при вызове этого метода. Фигурные скобки обязательны, даже если тело метода состоит из одного оператора.

При обращении к методу из этого же класса нужно указать только его имя.

```
<имя метода>(список параметров);
```

При обращении из другого класса необходимо указать полное имя.

```
<имя класса>.<имя метода>(список параметров);
```

В языке *C#* объявление методов и их реализация происходят одновременно, в других языках, в языках, например, *C++* и *Delphi* сначала идет описание класса, и лишь потом его реализация. В *C++* описание происходит в заголовочном файле, с расширением *.h*, а реализация в отдельном модуле с расширением *.cpp*.

В *C#* описание методов и их реализация находятся в одном файле, что делает код более компактным и более мобильным.

Объявление и описание метода в C# выглядит следующим образом:

```
<модификаторы> <тип возвращаемого значения> <имя метода> (список параметров через запятую)
{<код;>
[return <значение>;]
}
```

***Статические*** (или статичные) методы и переменные класса создаются один раз и связаны они именно с классом, а не с объектом. Сколько бы ни было объектов статическая переменная, если она есть в классе, одна.

Статический (статичный) метод может быть вызван без создания объекта.

Для его объявления нужно указать ключевое слово *static* при объявлении метода.

***Конструктор***

Это специфический метод класса, предназначенный для задачи начальных свойств объекта.

Имя конструктора совпадает с именем класса.

При описании конструктора, тип возвращаемого значения не указывают даже `void`.

Модификатор доступа класса указывает, может ли конструктор вызываться для класса извне.

```
public KV (int a)
{storona= a;}
…
KV kv1 = new KV(7);
```

Если конструктор не описать, в классе всегда будет существовать конструктор по умолчанию. Именно он и вызывался в предыдущих примерах, поэтому и были необходимы круглые скобки после имени класса.

При описании своего конструктора, остается возможность вызова конструктора по умолчанию.

Так как конструктор - это метод, то его можно перегрузить необходимое число раз.

Таким образом можно создавать объекты с различными начальными значениями свойств.

Рассмотрим вариант перегрузки конструктора по умолчанию, задающего начальные значения

```
public KV ()
{storona=1;}
```

Далее приведено описание класса `Kvadrat`, содержащего одно поле и соответствующее свойство.

```
public partial class Form1 : Form
{
public Form1()
{
InitializeComponent();
}
public class Kvadrat
{
private float storona;
public float Storona
{
get { return storona; }
set { storona = value; }
}
public float Square()
```

```
{return storona * storona; }
}// class
Kvadrat kv1 = new Kvadrat();
Kvadrat kv2 = new Kvadrat();
private void button1_Click(object sender, EventArgs e)
{
float a; bool flag=true;
if (float.TryParse(textBox1.Text, out a))
kv1.Storona = a;
else { textBox1.Text = "error"; flag = false; }
if (float.TryParse(textBox2.Text, out a))
kv2.Storona = a;
else {textBox2.Text = "ошибка ввода";flag = false; }

if (flag){float R = kv2.Square() - kv1.Square();
label.Text = "R=" + R.ToString() + " кв. см";}
}
}
}
```

Ключевое слово *partial*, указывает, что объявление и определение класса разделены. Это удобно при создании проектов, над которыми работает несколько разработчиков. И используется при создании класса формы.

***Деструктор или метод завершения***

Класс может иметь только один метод завершения.

Методы завершения не могут быть унаследованы или перегружены.

Методы завершения невозможно вызвать. Они запускаются автоматически.

Метод завершения не принимает модификаторов и не имеет параметров.

Описание деструктора на языке C# не отличается от описания деструктора на языке *C++*.

Описание начинается с символа тильда (~), далее следует имя класса. Весь необходимый код по освобождению памяти будет сгенерирован автоматически.

Так как деструктор невозможно вызвать вручную, действия необходимые совершить при завершении работы с объектом, прописывают в методах *Close()* и *Dispose().*

***Перегрузка операторов***

Перегруженный оператор в языке программирования C# должен быть объявлен как статический открытый метод класса.

Так как такие методы не получают связь с объектом, то количество передаваемых операндов должно соответствовать назначению операнда (отличие от перегрузки операторов в языке *C++*). Два - для бинарных, один - для унарных.

Как минимум один параметр, передаваемы в перегруженный оператор должен быть типа соответствующего класса.

Ниже приведён пример, перегрузки оператора + в классе квадрат. Метод возвращает сумму площадей квадратов.

```
public static float operator +(Kvadrat kv, Kvadrat kv2)
{return (kv.Square() + kv2.Square());}
```

В языке программирования C# отсутствует разница при перегрузке постфиксной и префиксной форм операторов инкремента и декремента Т.е. операторы ++ и -- перегрузить нужно один раз. При этом работа будет корректной как при вызове оператора в постфиксной, так при вызове в префиксной форме. Это существенное отличие от перегрузки этих операторов в *C++*. Далее приведён пример перегрузки оператора ++ в классе квадрат.

```
public static Kvadrat operator ++(Kvadrat K)
{
return new Kvadrat { storona = K.storona + 1 };
}
```

***Наследование***

В *C#* возможно только одиночное наследование. При объявлении класса наследника необходимо после его имени через двоеточие указать имя родительского класса.

```
class < имя класса >: <имя родительского класса>
public class Rect : Kvadrat
{…
}
Rect R1 = new Rect();// - создание объекта класса наследника.
```

Класс *Rect* – прямоугольник унаследовал от класса *Kvadrat* свойство *Storona.* Для корректного описания прямоугольника необходимо ввести описание ещё одной стороны и переопределить метод вычисления площади, предварив его ключевым словом *new.*

```
public class Rect : Kvadrat
{private float storona_2;
public float Storona_2
{
get { return storona_2; }
set { storona_2 = value; }
}
public new float Square()
{ return storona * storona_2; }
}
}
```

Чтобы было возможно в классе наследнике обратиться к полю `storona`, в базовом классе это поле должно быть объявлено с модификатором `protected`.

```
protected float storona;
```

Теперь можно задавать значения сторон и вычислять площадь прямоугольника.

```
R1.Storona =3;
R1.Storona_2 =4;
label1.Text = R1.Square().ToString();
```

Для обращения к скрытым членам в классе наследнике, это могут быть и поля, и методы, используют ключевое слово *base*. Его можно рассматривать как ссылку на базовый класс того класса в котором, *base* используется.

```
<base>.<член класса>
```

***Полиморфизм***

Объекты в языке программирования C# относятся к ссылочным типам *(reference type).*

Так как в C# все классы являются наследниками класса *Object*, можно объявить переменную этого класса и присвоить ей переменную, любого в том числе и пользовательского класса.

```
Object K= new Kvadrat();
((Kvadrat)K).Storona = 5;
```

```
label1.Text=((Kvadrat)K).Storona.ToString();
```

При работе с экземпляром класса наследника через ссылку на родительский класс будет вызван метод родительского класса, если он переопределён с ключевым словом *new*. В случае если нужно вызвать метод класса наследника, его следует переопределить с ключевым словом *override*, а в базовом классе такой метод должен быть объявлен с ключевым словом *virtual*.

И так, полиморфизм проявляется при работе с виртуальными методами, перегруженными в классе наследнике с ключевым словом *override*.

***Контрольные вопросы***

1. С какого ключевого слова начинается описание класса?
2. К какой категории типов данных относя объекты в языке программирования C#?
3. Какой вид наследования определен в языке программирования C#, множественное наследование, одиночное наследование, существуют оба вида наследования?
4. С каким модификатором должно быть объявлено поле–член в базовом классе, чтобы данные были скрыты от пользователя, но сохранилась возможность использования этого поля в классе наследнике?
5. С каким именем объявляют конструктор класса?
6. Каким образом происходит вызов деструктора в языке программирования C#? Сколько можно описать деструкторов?
7. В каком методе нужно прописать действия, которые необходимо выполнить при уничтожении объекта?
8. Какое ключевое слово применяется для перекрытия обычных (не виртуальных) методов и свойств класса наследника?
9. С каким ключевым словом нужно определить виртуальный метод или свойство в классе наследнике при необходимости его переопределить?
10. Каково назначение ключевого слова *base*?

***Задания для самостоятельной работы***

1) Реализовать на языке программирования C# задания для самостоятельной работы, приведённые после первой – третьей лабораторных работ.

2) Описать класс для работы с комплексными числами, методы класса должны позволять реализовать все допустимые для комплексных чисел операции.

3) Описать класс вектор, задаваемый координатами на плоскости, методы класса должны обеспечивать допустимые для векторов операции.

4) Описать класс наследник класса вектор из предыдущей задачи, новый класс должен описывать трёхмерный вектор.

5) Описать класс прямоугольник со сторонами параллельными осям координат, разработать методы перемещения прямоугольника на плоскости, масштабирования.

6) Описать класс многочленов от одной переменной, задаваемых степенью многочлена и набором коэффициентов. Описать методы вычисления значения многочлена для заданного аргумента, сложения, вычитания и умножения на число.

7) Описать класс – «Матрица» с возможностью изменения количества строк и столбцов, вывода на экран всей матрицы и матрицы заданного размера.

8) Описать класс «Записная книжка» с возможностью работы с произвольным числом записей, поиска записи по какому-либо признаку (например, фамилии, дате рождения и т.п.)

9) Описать класс «Студенческая группа». Организовать возможность работы с переменным числом студентов, добавления удаления записей, поиска по какому-либо ключевому полю.

10) Описать класс «Домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска по какому-либо ключевому полю.

## Разработка визуального приложения на C#. Работа с формой

После создания визуального приложения решение будет содержать два модуля `Program.cs` и `Form.cs`. Структура проекта представлена на рисунке 14.

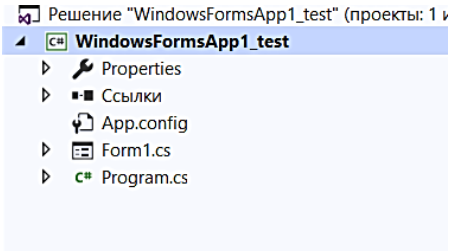



Рис. 14. Структура созданного проекта

В файле `Program.cs` содержится метод `Main()`. Этот метод вызывается первым при запуске приложения.

```
namespace WindowsFormsApp1_test
{
static class Program
{
[STAThread]
static void Main()
{
Application.EnableVisualStyles();
Application.SetCompatibleTextRenderingDefault(false);
Application.Run(new Form1());
}
}
}
```

`[STAThread]` – указывает, что модель потоков для приложения будет одиночная. Такой атрибут должен быть указан для всех точек входа в Windows Forms – приложения.

Строки кода метода `Main()` – обращение к классу *Application*, содержащему свойства и методы для поддержки работы приложения. Эти свойства и методы статичные, т.е. для их вызова не нужно создавать объект.

Метод `EnableVisualStyles()` включает отображение в стилях ОС на которой запущено приложение.

Метод `Application.SetCompatibleTextRenderingDefault(false);` если в качестве параметра передать *false*, позволяет использовать современные средства отображения текста.

Метод `Application.Run(new Form1());` получает в качестве параметра экземпляр класса формы, отображает форму и запускает цикл обработки сообщений ОС.

Двойной щелчок по файлу `Form.cs` открывает режим визуального дизайна, в котором можно расставлять компоненты на форме.

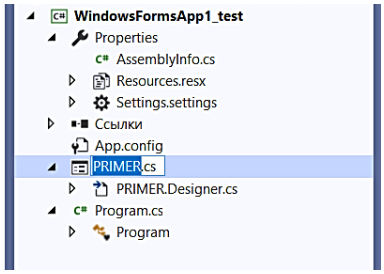
***Именование формы***

Так как все имена должны быть информативными, проект не должен содержать имени `Form1`. Поэтому форме нужно дать осмысленное информативное имя.

Для этого нужно дважды щёлкнуть правой кнопкой мыши по имени файла формы в окне обозревателя решения.

Выбрать пункт «*Переименовать*» и задать новое имя. При этом будут изменены имена не только файлов, но и класса, описывающего форму.

Изменения затронут все файлы, в которых задействована форма. Результат представлен на рисунке 15.




а)

```
Application.Run(new PRIMER());
```

б)

Рис. 15. Изменение имени формы
а) выбор файла, б) результат изменений

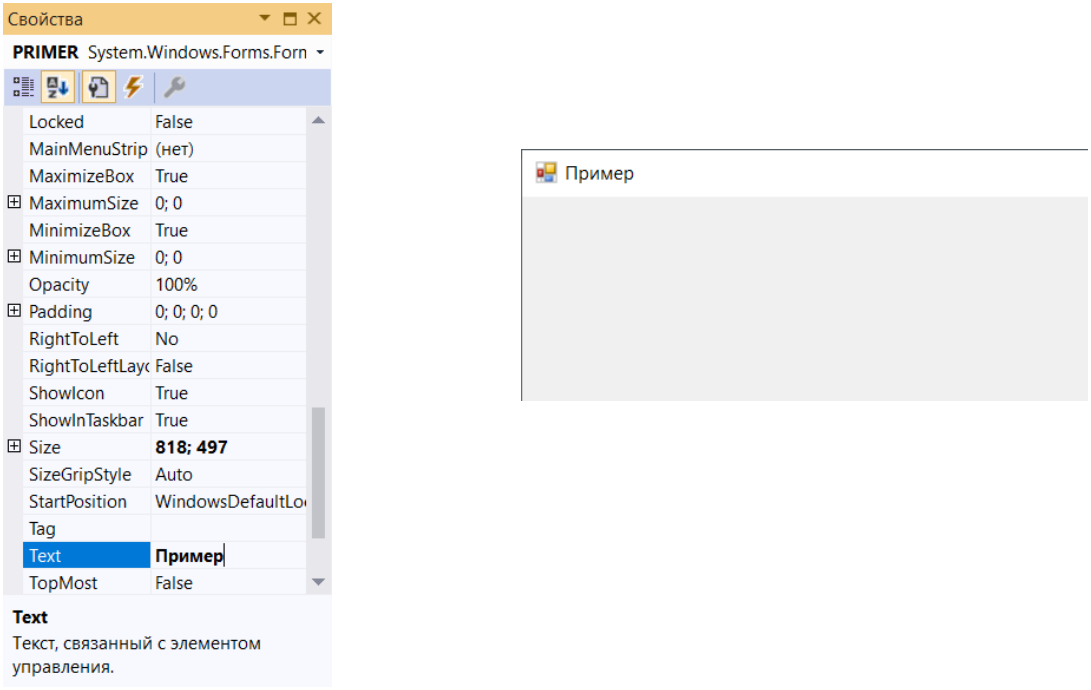
***Изменение свойств формы***

Свойства формы, как и размещённых на ней компонентов, можно изменять программно (в коде) и в окне редактора свойств.

Первый способ используют если изменения должны произойти во время работы программы, второй для задания свойств во время разработки приложения.

Изменим заголовок формы.

Текст заголовка содержится в свойстве *Text*. Процесс и результат изменения значения свойства представлен на рисунке 16.




а) б)

Рис. 16. Изменение значения свойства *Text*

а) процесс изменение свойства б) результат изменения свойства

Разместим на форме компонент *Label*, рисунок 17. Компонент *Label* предназначен для отображения текста. Далее разместим на форме компонент *Button* – кнопку и напишем обработчик события нажатия на эту кнопку.


а) б)

Рис. 17. Добавление компонента на форму

а) – выбор компонента; б) результат добавления

Программное изменим текст метки.

Отображаемый текст, как было сказано ранее, хранится в свойстве *Text*.

Изменения нужно внести в конструктор формы, модуля PRIMER.cs

```
namespace WindowsFormsApp1_test
{
public partial class PRIMER : Form
{
public PRIMER()
{
InitializeComponent();
label1.Text = "Это Метка";
}
}
}
```

На рисунке 18 показано изменение отображаемого текста, содержащегося в свойстве *Text*, после запуска приложения.

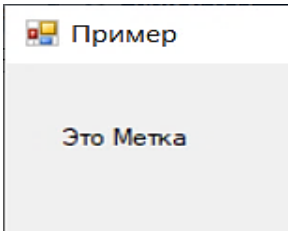
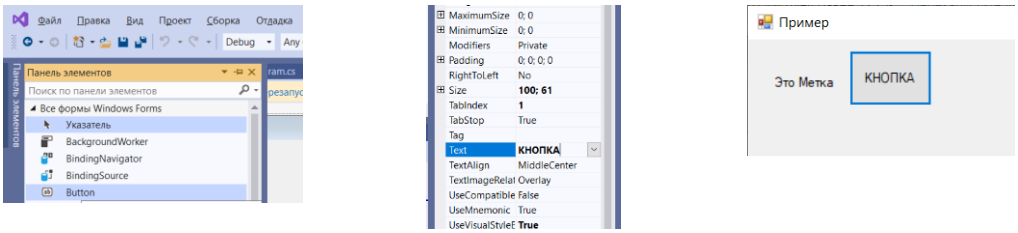



Рис. 18. Изменение текста метки

Добавляем кнопку, изменяем текст на ней. Последовательность действий, проиллюстрирована на рисунке 19.




*а)* *б)* *в)*

Рис. 19. Добавление на форму кнопки и изменение её свойства *Text*

а)выбор компонента б) изменение свойства в) результат

Двойной щелчок по кнопке в режиме проектирования, добавляет код обработчика события (в нашем случае это нажатие на кнопку).

```
namespace WindowsFormsApp1_test
{
public partial class PRIMER : Form
{
public PRIMER()
{
InitializeComponent();
label1.Text = "Это Метка";
}
private void Button1_Click(object sender, EventArgs e)
{
/*между этими фигурными скобками необходимо писать код обработчика события Button1_Click */
}
}
}
```

## Компоненты. Свойства. Методы. События

***Разработка визуального приложения***

Работа по разработке визуального приложения сводится к размещению на форме компонент и написания обработчиков событий.

Компоненты можно разделить на два вида визуальные и невизуальные. Визуальные видны на форме во время проектирования в том же виде, что и во время работы приложения. Невизуальные отображаются во время проектирования в виде пиктограмм. А во время исполнения программы не видны до определенного момента.

Свойства объекта определяют его внешний вид и некоторые моменты поведения. Например, цвет размер, возможность изменять значение в процессе работы приложения.

После размещения компонент на форме возможно изменение значений свойств, заданных по умолчанию.

Изменение свойств возможно двумя способами.

1) Изменение значения в окне редактирования свойств.

2) Программное изменение. Происходит на шаге исполнения приложения.

Для этого необходимо в левой части оператора присваивания указать имя объекта, далее точку и имя свойства.

*<имя объекта>.<имя свойства>*

Пример.

```
label1.Text= “метка”;
```

В том же окне есть закладка *События*.

У каждого компонента существует ряд событий, обработчики, которых можно написать.

Событие происходит во время выполнения программы, например, нажатие клавиши на клавиатуре или щелчок кнопкой мыши.

Если написан код обработчика события, после наступления события, управление передаётся первому оператору обработчика.

Двойной щелчок по компоненту во время проектирования приводит к добавлению в проект обработчика *основного события* и переходу на страницу редактирования кода.

Например, для кнопки это событие – `Click`.

Имя объекта доступно для редактирования и содержится в свойстве `Name.` По умолчанию объектам дается имя, состоящее из двух частей. Первая это имя компонента с маленькой буквы, вторая порядковый номер объекта соответствующего типа. Например, `label1, label2.`

Всем объектам стоит давать осмысленные имена, т.е. первую часть оставляют, а порядковый номер заменяют на информативную часть, говорящую о роли объекта в приложении.

Например, `labelName`.

Метка с таким именем может располагаться перед окном, в котором пользователь должен будет ввести свой имя.

Имя объекта необходимо для того, чтобы иметь возможность к нему обратиться, текст надписи, отображаемой на компоненте содержится в свойстве `Text`.

Ряд свойств имеют свои свойства, их можно увидеть в списке, раскрывающемся при выборе данного свойства. Например, свойство `Font` – является самостоятельным классом со своими свойствами.

В окне *Properties* (*Свойства*) отображаются свойства активного компонента.

***Основные свойства формы***

`BackColor` – цвет фона окна.

`BackgroundImage` – изображение на форме.

`Cursor` – позволяет выбрать вид, которым будет отображён курсор, при наведении на окно.

`Font` – шрифт, которым будет отображён цвет поверх окна.

`ForeColor` – цвет переднего плана, цвет текста.

`Icon` – иконка формы.

`Name` – имя формы, которое можно использовать в коде.

`Opacity` – прозрачность окна в процентах, по умолчанию 100%, окно абсолютно непрозрачно.

`ShowIcon` – свойство имеет логическое значение, определяющее нужно ли отображать иконку в заголовке окна

`Size` – размер формы, как и любого окна можно изменить путем перемещения его границы. При этом изменятся свойства `Height` и `Width` свойства `Size`.

`Tag` – в этом свойстве можно сохранить любую информацию т.к. свойство имеет тип `Object`.

`Text` – заголовок окна.

`WindowsState` – состояние окна (можно выбрать одно из трех значений: `Normal, Maximized, Minimized`)

***Методы формы***

`Show()` – отобразить окно немодально, т.е. не блокируя работу родительского окна.

`Hide()` – спрятать окно, не уничтожая его.

`Close()` – закрыть, при этом уничтожается объект формы.

`Invalidate()` – перерисовать форму.

***Общие компоненты или стандартные элементы управления***

Далее приведены назначения свойств, которыми обладают все визуальные компоненты.

`Anchor` – позволяет определить к какой стороне формы прикрепить компонент.

`BackColor` – цвет фона.

`Cursor` – задание формы курсора, которую он принимает при наведении на объект.

`Dock` – выравнивание.

`Enabled` – управляет возможностью доступа к компоненту, принимает значения `true/false`. По умолчанию имеет значение `true` – компонент доступен, при задании значения *false*, компонент становится не доступным, но видимым, отображается светло серым.

`Font` – задает свойства шрифта, которым будет отображён текст на компоненте.

`ForeColor` – цвет переднего плана (текста).

`Location` – положение компонента относительно левого верхнего угла родительского объекта, в простейшем случае это форма.

`MaximumSize` – максимальный размер элементов управления.

`MinimumSize` – минимальный размер элементов управления.

`Name` – имя компонента, используемое в коде (не текст).

`Padding` – отступы текста от всех сторон компонента.

`Size` – размер компонента, задан значениями высоты и ширины `(Height, Width)`.

`Visible` – управляет видимостью компонента во время выполнения программы, имеет значения `true/false`, значение по умолчанию – `true,` компонент виден, при изменении значения на `false` компонент не будет удален, а будет невидим, до изменения значения этого свойства на `true`.

Ниже приведён обзор наиболее часто используемых компонент. При этом рассматриваются их важные свойства, методы и события. Если компонент обладает свойствами, описанными у компонента, рассмотренного ранее, обзор свойств не дублируется.

Добавить обработчик основного событиям можно двойным щелчком по компоненту во время проектирования. Для создания обработчика другого события, определённого для соответствующего компонента, нужно выбрать это событие в списке событий выбранного компонента и также выполнить двойной щелчок по нужной строчке.

При этом в правом столбце будет отображено имя обработчика, а в соответствующий файл добавлен код, содержащий заголовочную строку и пустые фигурные скобки, в которых нужно писать тело метода.

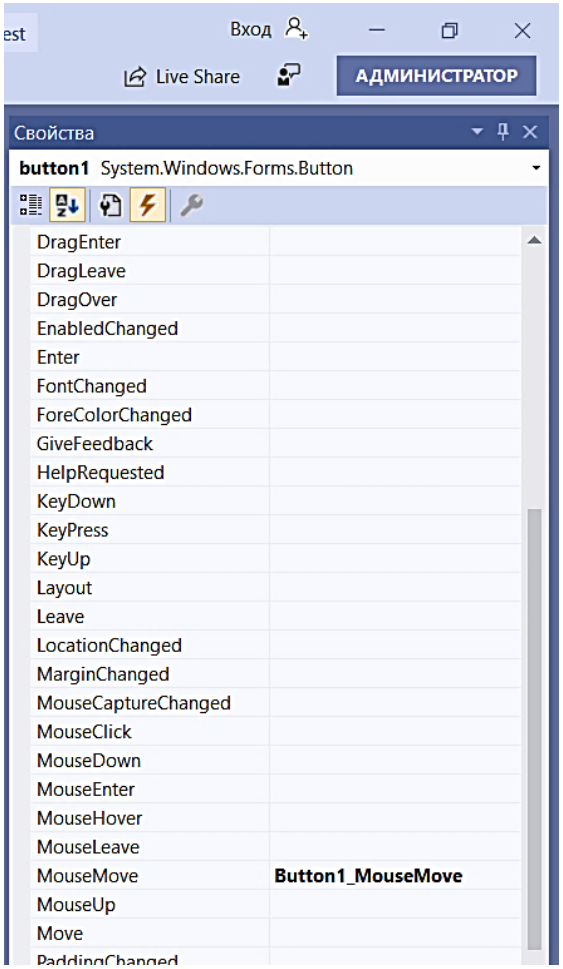



Рис. 20. Выбор события *MouseMove* для компонента *Button1*

```
    private void Button1_MouseMove(object sender,
MouseEventArgs e)
    {
    }
```

Для отображения текста на поверхности формы, например, обозначения темы, заголовка, поясняющей надписи к значению какого-либо другого объекта, чаще всего используют компонент ***Label*** - метка.

***Основные свойства компонента Label***

`Text` – текст, отображаемый на поверхности компонента.

AutoSize - по умолчанию имеет значение *true*, компонент автоматически принимает минимально необходимые размеры для отображения текста.

TextAlign - указывает сторону по которой выравнен текст, использование этого свойства бессмысленно при AutoSize равном true.

Font – параметры шрифта, используемого для отображения текста.

Компонент ***TextBox*** позволяет пользователю вводить текст.

***Основные свойства и события компонента TextBox***

Text – имеет строковый тип, по умолчанию пустая строка.

Multiline – разрешает многострочное отображение данных.

Modified - значение свойства изменяется на true если пользователь внёс изменения в строку, сбросить значение на false, можно только вручную.

UseSistemPassword – вводимые символы будут скрыты за точками, как при вводе пароля.

TextChanged – событие генерируется при изменении текста пользователем.

Выпадающий список компонент ***ComboBox***. Этот компонент удобен при необходимости выбрать одно значение из списка.

***Основные свойства и события компонента ComboBox***

Items – в этом свойстве задаются элементы списка, свойство представляет собой коллекцию строк.

MaxDropDownItems – количество строк, видимых при раскрытии списка.

SelectedIndex – свойство, возвращающее номер выделенной строки, если ничего не выделено значение свойства равно -1.

SelectedItem – текст выделенной строчки, если ничего не выделено значение null.

Перед использованием значения свойства, его необходимо проверить на null.

Иначе использование метода ToString() вернёт ошибку.

Пример такой проверки.

```
if (comboBox1.SelectedItem !=null)
{string str = comxboBox1.SelectedItem.ToString();
MessageBox.Show(str);
```

```
}
else
MessageBox.Show(“нет выделения”);
```

Add() – метод добавления строки в коллекцию.

AddRange() - метод, позволяющий добавлять сразу множество элементов в коллекцию строк, методу необходимо передать массив строк.

Remove() – метод удаления строки из коллекции.

Clear – метод очистки списка коллекции.

```
comboBox1.Items.Clear();
```

Наиболее интересное событие SelectedIndexChange() – изменение индекса выделенной строки.

Компонент ***CheckBox*** позволяет выбрать одно из двух состояний. Используется, когда пользователь должен дать ответ на вопрос «Да» или «Нет».

***Основные свойства компонента CheckBox***

Text – текст, поясняющий значение флажка.

Checked – имеет значение true, если флажок установлен иначе - *false*.

AutoCheck – по умолчанию true, это значит, что значение свойства checked изменяется при щелчке на изображении флажка.

TextAllign – выравнивание текста в компоненте.

CheckAllign – положение кнопки в компоненте;

События компонента CheckBox.

Click() – генерируется при щелчке по компоненту.

CheckedChange() – генерируется при изменении свойства Checked.

Компонент ***CheckedListBox*** - список с индикаторами. Представляет из себя список, элементы, которого хранятся в свойстве Items, имеющем тип коллекции.

***Основные свойства компонента CheckedListBox***

Items – строки списка.

MultiColumn – значение true разрешает отображать строки в несколько колонок. При большом количестве строк имеет смысл установить свойство MultiColumn в значение true.

`SelectedItem` – свойство, содержащее выбранный элемент списка, не отмеченный галочкой, их может быть много, а тот элемент, который выделен в данный момент, если ничего не выделено свойство равно `null`.

`CheckedItems` – содержит список элементов, отмеченных галочками.

Коллекция `CheckedItems` имеет следующие основные методы

`Add()` – добавление;

`Remove()` – удаление;

`Clear()` – очистка;

```
checkedListBox1.Items.Add("new line", true);
```

***Основные события компонента CheckedListBox***

`SelectedIndexChanged` – срабатывает при изменении индекса выделенного элемента.

`SelectedValueChanged` – срабатывает при изменении выделенного значения.

Выпадающий список ***ListBox*** похож на предыдущий компонент. За исключением отсутствия флажков индикаторов.

***Свойства компонента ListBox***

`SelectedItem` – значение текущего выбранного элемента

`SelectionMode` – разрешает или запрещает множественный выбор.

`SelectedItem` – свойство, содержащее выбранный элемент списка.

`SelectedItems` – тоже самое если разрешён множественный выбор.

Пример перебора выделенных строк и вывод их в окне.

```
foreach (string str in listBox1.SelectedItems)
MessageBox.Show(str);
```

Наиболее часто используемые события компонента ListBox.

`SelectedIndexChanged` – срабатывает при изменении индекса выделенного элемента.

`SelectedValueChanged` – срабатывает при изменении выделенного значения

Компонент ***DateTimePicker*** позволяет пользователю водить дату и время.

`Value` – свойство, хранящее выбранную пользователем дату, имеет тип `DateTime`.

`ValueChanged` – событие, генерируется каждый раз при изменении даты или времени внутри компонента.

Компонент ***PictureBox*** используют при необходимости добавить изображение на форму, например, фотографию при заполнении анкеты.

Основное свойство компонента – `Image`.

`SizeMode` - режим отображения выбранной картинки. Это перечисление, позволяющее выбрать одно из следующих значений:

`StretchImage` – растянуть картинку поверх всей поверхности компонента.

`AutoSize` – компонент автоматически примет размеры картинки, чтобы она была видна полностью.

`CenterImage` – отцентрировать картинку внутри компонента.

`Zoom` – уменьшить картинку так чтобы она вписывалась в компонент, при этом пропорции изображения сохраняются.

`Normal` – просто отобразить картинку, не растягивая ее.

Компонент ***ProgressBar*** позволяет показать пользователю процент выполненных действий.

Имеет три важных свойства.

`Maximum` – максимальное значение индикатора, по умолчанию равно 100.

`Minimum` – минимальное значение индикатора, по умолчанию равно 0.

`Value` – текущее значение.

Компонент ***RadioButton*** позволяет выбрать пользователю одно из двух состояний.

Имеет смысл размещать несколько таких компонентов в контейнере. При этом только один из них может быть выделен. Под контейнером понимается любой компонент, на поверхности которого можно расположить другие компоненты.

***Основные свойства компонента RadioButton***

`Text` – содержит текст, отображенный около индикатора.

`Checked` – указывает выделен компонент или нет.

`AutoCheck` – если установить в `false`, то при щелчке по этому компоненту, он автоматически не выделится. Выделение можно сделать в обработчике событие `Click` на этом компоненте.

Компонент ***Button*** – кнопка.

Основное свойство `Text` – содержит текст, отображаемый на кнопке.

Основное событие `Click` – срабатывает при щелчке по кнопке вовремя выполнения программы.

Чтобы перейти к написанию обработчика основного события необходимо дважды щёлкнуть мышкой по компоненту во время разработки.

Компонент ***CheckBox*** позволяет выбрать одно из двух состояний. Используется при необходимости получения от пользователя ответа: «да» или «нет».

***Основные свойства компонента CheckBox***

`Checked` – имеет значение *true* если значение выбрано, то есть в компоненте стоит флажок, иначе свойство имеет значение *false*.

`CheckState` – состояние в виде перечисления, в котором можно указать одно из трёх значений.

***Основные события компонента CheckBox***

`CheckedChanged` – наступает при изменении состояния `Checked.`

`CheckedStateChanged` – генерируется при изменении значения свойства `CheckState`.

`Click` – срабатывает при щелчке по компоненту.

***Компоненты контейнеры***

Это компоненты на поверхности, которых можно размещать, группировать другие компоненты. Используют для визуального объединения управляющих компонентов, предназначенных для решения общей задачи.

Компонент ***Group Box***

Если на поверхности этого компонента разместить несколько компонентов `RadioButton`, то они будут работать в связке, т.е. выбранным в один момент времени может быть только один компонент, при выборе другого, значение свойства `Checked` предыдущего выбранного компонента становится равно `false`.

Свойство `Text` содержит текст заголовка группы.

Компонент ***Panel*** предназначен для размещения объектов. Для удобства компоновки элементов, работу с панелью обычно начинают с задания свойства `Dock`, указывающего вдоль какого края формы будет размещена панель. При тематической группировке элементов, есть

смысл размещать на форме несколько панелей, разделяя их компонентом Splitter.

Компонент ***TabControl***

Представляет собой набор страниц, с заголовками, между которыми можно переключаться.

***Свойства страниц TabPage***

Text - содержит заголовок страницы.

ToolTopText - содержит текст подсказки для страницы, что бы подсказка отображалась при наведении курсора на заголовок страницы, необходимо задать значение свойства ShowToolTips – true.

***Свойства компонента TabControl***

TabPages – коллекция страниц.

TabIndex - индекс текущей страницы.

HotTrack – страницы будут подсвечиваться при наведении на них курсора, если значение свойства true.

Multiline – разрешение отображать заголовок страниц в несколько строк при необходимости.

## *Лабораторная работа № 5. Разработка визуального приложения*

***Цель работы.***

Совершенствование навыков объектно-ориентированного программирования.

Получение навыков работы в визуальной среде.

Знакомство с визуальными компонентами. Получение навыков разработки простейших обработчиков событий.

***Постановка задачи.***

Используя изученные компоненты разработать приложение, позволяющие решить приведенные ниже три задачи.

Задачи оформить в виде одного оконного приложения, использовав для этого компонент *TabControl*. Решение каждой задачи разместить на отдельной странице компонента. Организовать всплывающие подсказки для каждой страницы. Формулировку каждой задачи отобразить на странице, реализующей её решение.

Для ввода данных использовать компоненты *TextBox*, для сопровождающих надписей компоненты *Label*, вычисления производить в обработчике события нажатий на кнопку компонента Button.

1) Имеется серия измерений элементов треугольника. Группы элементов пронумерованы. В серии в произвольном порядке могут встречаться такие группы элементов треугольника: основание и высота; две стороны и угол между ними (угол задан в радианах); три стороны. Разработать программу, которая запрашивает номер группы элементов, вводит соответствующие элементы и вычисляет площадь треугольника. На форме разместить компонент *RadioButtonList*, содержащий три строки, соответствующие трем группам измерений.

2) Начав тренировки, спортсмен в первый день пробежал $S$ км. Каждый день он увеличивал дневную норму на 10% нормы предыдущего дня. Какой суммарный путь пробежит спортсмен за $N$ дней? $S$ и $N$ входные параметры, задаваемые пользователем в соответствующих окнах ввода.

3) Одноклеточная амеба каждые 3 часа делится на 2 клетки. Определить, сколько амеб будет через 3, 6, 9, 12,…, 24,… часа. Начальное количество амёб и необходимое число часов задать входными параметрами.

***Контрольные вопросы***

1. Какие два способа изменения значения свойства компонента существуют?
2. Каково назначение компонента *label*?
3. Каково назначение свойства *Text*?
4. Каково назначение компонент контейнеров?
5. Перечислите известные вам компоненты контейнеры?
6. Каким образом можно организовать добавление в программу автоматически сгенерированного кода обработчика (заголовок и пустые фигурные скобки)?
7. Какие компоненты можно использовать, для организации получения ответа от пользователя да/нет?
8. Поясните назначение компонента *Button*.
9. Назовите основное событие компонента из предыдущего вопроса?
10. Какие компоненты позволяют организовать выбор пользователем одного или нескольких значений из списка?

***Задания для самостоятельной работы***

1) Около стены наклонно стоит палка длиной $x$ м. Один ее конец находится на расстоянии $y$ м от стены. Определить значение угла

$\alpha$ между палкой и полом для значений если $x$ задано пользователем, а $y$ изменяется от 2 до 3 м с шагом $h$ м, также задаваемым пользователем.

2) У гусей и кроликов вместе 64 лапы. Сколько может быть кроликов и гусей (отобразить на форме все возможные сочетания)?

3) Написать программу расчета суммы вклада в банке с учетом процентной ставки. Сумма вклада на год рассчитывается следующим образом. Сумма=Сумма*Ставка/100. Сумма вклада на $n$ лет рассчитывается следующим образом. Сумма=$\sum_{i=0}^{n}$ Сумма $*$ Ставка/100.

4) Написать программу расчета стоимости поездки на автомобиле. Входные данные (поля заполняются пользователем): цена литра бензина, потребление бензина (литр на 100 км), расстояние. Написать программу для расчета сопротивления резисторов, соединенных параллельно или последовательно. Выбор соединения организовать, используя компонент *RadioButtonList*, для ввода данных использовать компоненты *TextBox*, на форме должны быть указана единицы измерения.

5) Написать программу конвертер для перевода единиц измерения длинны. Входные данные: значение в исходных единицах измерения, исходные единицы измерения, результирующие единицы измерения. Расчет производить в обработчике события *Click* объекта *Button*.

6) Написать программу для решения квадратного уравнения вида $Ax^2+Bx+C=0$. Предусмотреть поля ввода коэффициентов и кнопки вывода справочной информации и решения.

*7)* Написать программу «Секундомер». Предусмотреть поля для отображения текущего времени, отображения интервала смены показаний текущего времени. Предусмотреть кнопки пуска и останова секундомера, а также сброса текущего времени. Для управления секундомером использовать стандартный невизуальный компонент *Timer*.

8) Написать программу, для создания списка преподавателей и студентов. Предусмотреть поля ввода атрибутов: фамилии, группы или кафедры, а также выбор с помощью статуса вносимого в список (студент или преподаватель), кнопки для внесения в список и отображения списка студентов или преподавателей. Описать базовый класс *Person*, и два класса наследника, *Teacher* и *Student*. Для размещения сведений о студентах и преподавателях использовать единый массив.

9) Составить расписание на неделю. Пользователь вводит порядковый номер дня недели после чего на форме отображается, расписание на день.

10) Написать программу «Восточный календарь», пользователь вводит год, на форме отображается название животного, символизирующего этот год по восточному календарю.

## *Лабораторная работа № 6. Организация выбора пользователя из двух и множества вариантов*

***Цель работы.*** Совершенствование навыка работы студентов в визуальной среде. Дальнейшее изучение возможностей свойств и методов базовых компонентов.

***Постановка задачи.*** Написать программу расчета стоимости заказа по электронному меню.

Пользователю предложить выбор из некоторого списка с возможностью выбора строк.

Список организовать, использовав компонент *CheckListBox*, строки, которого отображают название и цену блюда.

Цены хранить в одномерном числовом массиве, с сохранением соответствия индексов массива цен и списка блюд.

При выборе пользователем строки меню или отмене выбора стоимость заказа должна быть пересчитана.

Организовать возможность задания пользователем различного количества для каждой позиций заказа. Возможность редактировать количество становится доступной при выборе отмеченной позиции.

***Контрольные вопросы***

1. Какие компоненты позволяют сделать пользователю выбор одного или нескольких вариантов из списка?
2. Какой метод коллекции позволяет добавить строку в коллекцию?
3. Какой метод позволяет добавить несколько строк в коллекцию?
4. Назовите метод удаления строки из коллекции?
5. Назовите метод очистки списка строк?
6. Каким образом можно перебрать коллекцию всех строк компонента *CheckListBox*?

7. Каким образом можно перебрать коллекцию всех выбранных строк компонента *CheckListBox*?

8. Каким образом можно проверить выбрана ли определенная строка компонента *CheckListBox?*

9. Каким образом можно получить значение выделенной строки компонента *CheckListBox?*

10. Каким образом можно получить номер выделенной строки компонента *CheckListBox?*

***Задания для самостоятельной работы***

1) Написать программу, предоставляющую пользователю возможность выбрать несколько иностранных языков для изучения из предложенного списка.

2) Написать программу, позволяющую скопировать выбранные в компоненте *CheckListBox* строки в текстовое поле любого предназначенного для отображения текста компонента.

3) Написать обработчик события нажатия на кнопку, инвертирующий выбор строк в компоненте *CheckListBox.*

4) Написать обработчик события нажатия на кнопку позволяющий добавить в компонент *CheckListBox* новую строку, введённую в текстовом в поле компонента *TextBox.*

5) Написать программу, позволяющую выбрать из списка с флажками значения, скопировать их в компонент *ListBox* в алфавитном порядке.

6) Написать программу, позволяющую выбрать из списка с флажками значения и отобразить их на форме любым способом, удалив при этом выбранные строки из списка.

7) Написать обработчик события *SelectedIndexChanged* компонента *CheckListBox* сохраняющий текст всех выделенных строк в файл.

8) Создать визуальное приложение, позволяющее организовать при помощи компонента *CheckListBox* выбор пользователем предметов для дополнительного изучения, предметов должно быть не меньше двух и не больше четырёх.

9) Добавить в компонент *CheckListBox* строки, из выбранного файла, при этом в состояние *Checked – true,* установить только значение первой строки.

10) Создать визуальное приложение, позволяющее организовать при помощи компонента *CheckListBox* выбор пользователем видов досуговой деятельности, выбранные значения сохранить в текстовый файл.

## *Лабораторная работа № 7. Разработка обработчиков событий. Калькулятор*

***Цель работы.*** Совершенствование навыка самостоятельной разработки обработчиков событий.

***Постановка задачи.*** Разработать приложение «Калькулятор», реализующее основные арифметические действия +,-,*,/, унарный минус.

Реализовать возможность удаления последнего символа из вводимой строки (значение операнда).

Предусмотреть возможность повторного нажатия на равно. При этом должно происходить вычисление последней операции с использованием последнего результата в качестве первого операнда, значение второго операнда остаётся прежним. Например, 5+3=8=11=14=17 и т.д.

Реализовать повторное нажатие на знак действия и сразу за ним знак равно. При этом происходит выполнение операции с использованием последнего результата в качестве первого и второго операндов. Например, 3+4=7+=14-=0.

Предусмотреть вывод соответствующего сообщения в случае попытки деления на ноль.

***Контрольные вопросы***

1. Какое событие для компонента называют основным?
2. Какое событие является основным для компонента *Button?*
3. Каким образом можно добавить в код обработчик события основного для компонента?
4. Каким образом, можно, не дублируя код, организовать выполнение действий по одному алгоритму, но с разным параметром при нажатии на разные кнопки?
5. Какие действия необходимо выполнить в коде при работе с вещественными числами, если пользователю предоставлена возможность вводить эти данные в текстовое поле?
6. Какие действие нужно выполнить, чтобы отобразить на форме, например, с помощью компонента *Label,* значение квадратного корня из вещественного числа, число вводит пользователь в поле для ввода компонента *TextBox,* приведите фрагмент кода?
7. Назовите основное событие компонента *CheckBox.*

8. Напишите обработчик события, позволяющий скрыть или отобразить панель и всю расположенную не ней информацию при изменении значения свойства *Checked* компонента *CheckBox.*

9. Напишите обработчик события *CheckedChanged* компонента *RadioButton* позволяющий рассчитать значение параметра *R*, алгоритм расчёта зависит от выбора одного их четырёх компонентов *RadioButton,* размещённых в одном контейнере.

10. Какой компонент предпочтительнее использовать если нужно предоставить пользователю возможность указания целого числа из некого диапазона?

***Задания для самостоятельной работы***

1) Создайте приложение «Калькулятор комплексной арифметики». Реализовать возможность ввода операндов, отображение и сохранение в первом операнде результата вычислений, суммы двух комплексных чисел, разности, умножения, деления,

2) Создайте приложение «Калькулятор векторной алгебры». Реализовать возможность ввода операндов, отображение результата вычислений, суммы двух векторов, разности, умножения, умножения на скаляр.

3) Создайте приложение «Тригонометрия». Представляющего собой тренажёр для запоминания значений синуса и косинуса углов (0°,30°,45°,60°,90°,180°).

4) Создайте приложение «Закон Ома для участка цепи». Предусмотрев выбор искомого параметра и задание входных значений известных параметров.

5) Создайте приложение «Таблица умножения». Написать программу-тренажёр для запоминания таблицы умножения, с проверкой введённого пользователем результата.

6) Создайте приложение «Меню». Предусмотреть выбор категории блюд, блюд из категории с указанием их количества, и расчёт полной стоимости.

7) Создайте приложение «Матрица». Реализовать выбор размерности и ввод значений, сложение, умножение, транспонирование и другие операции

8) Создайте приложение «Конвертер». Реализовать ввод пользователем значения, выбор единиц измерения из определённой категории величины, входных и итоговых.

9) Создайте приложение «Электронные весы». Позволяющее рассчитать калорийность блюда по выбранным из списка ингредиентам и из весу.

10) Создайте приложение «Расписание». По введенному пользователем дню недели, нужно отобразить расписание и домашнее задание на этот день.

## Работа с графикой

Основной класс для работы с графикой в *C#* - *Graphics*, класс находится в пространстве имен *System.Drawing*.

Класс Drawing реализует поверхность и методы рисования.

Для получения объекта этого класса используют обработчик события *Paint*.

Если создать для формы обработчик события *Paint*, будет сгенерирован следующий код:

```
private void Form1_Paint(object sender, PaintEventArgs e)
        {

        }
```

Второй параметр метода это переменная класса `PaintEventArgs`, через которую мы получаем:

`ClipRectangle` – область, которую нужно перерисовать;

`Graphics` – экземпляр класса `Graphics`, который представляет собой поверхность для рисования.

Начало координат поверхности (точка (0;0)) - левый верхний угол. Это необходимо учитывать при расчёте координат отображаемых графических объектов.

Таким образом, в обработчике события *Paint* для рисования на форме необходимо использовать объект *e.Graphics* и его методы.

В остальных методах объект этого класса нужно создавать самостоятельно.

При необходимости создать объект класса *Graphics* следует использовать метод *CreateGraphics* формы или элемента управления, на котором необходимо отобразить графику.

Если предполагается рисовать на форме, код будет таким:

*Graphics g = this.CreateGraphics();*

Если предполагается рисовать на заранее созданном контейнере. В данном случае это панель, то создание объекта класса *Graphic*s будет выглядеть так:

```
Graphics g = this.panel1.CreateGraphics();
```

***Основные методы класса Graphics***

`Clear( )` – очистить поверхность рисования и залить ее цветом, указанным в качестве параметра;

`DrawLine( )` – отображает линию;

`DrawRectangle( )` – рисует прямоугольник;

`DrawEllipse( )` – рисует эллипс;

`FillEllipse( ), FillRectangle( )` – залить цветом.

`DrawArc ( )` – рисует дугу;

***Примеры использования перечисленных выше методов***

```
g.Clear(Color.Red);
```

Заливка области для рисование красным цветом.

```
g.DrawLine(new Pen(Color.Green, 2),10,10,100,100);
```

Отображение на поверхности линии зелёного цвета, толщиной 2 пикселя, с координатами начальной точки (10;10) и координатами конечной точки (100;100).

```
g.DrawRectangle(new Pen(Color.DarkBlue, 3), 10, 10, 100, 100);
```

Отображение на поверхности прямоугольника синего цвета с координатами левого верхнего угла (10;10) и правого нижнего угла (100;100), толщина линии 3 пикселя,

```
g.DrawEllipse(new Pen(Color.Black, 1), 15, 15, 125, 125);
```

Отображение на поверхности эллипса черного цвета с координатами левого верхнего угла (15;15) и правого нижнего угла (125;125), толщина линии 2 пикселя.

Для того что бы иметь возможность задавать фигурам цвет, необходимо определить структуру типа *Color*.

Эта структура имеет только конструктор по умолчанию. Поля этой структуры изменять нельзя.

```
Color c = new Color();
```

После создания цвет изменить нельзя.

Его можно задать при помощи статических методов *Color*. Например метод `FromName()`, принимающий в качестве параметра название цвета в виде строки.

```
Color c =Color.FromName(“Green”);
```

Если необходим один из основных цветов, то используют статические свойства структуры *Color*. Для основных цветов есть предопределенные значения.

```
Color c =Color.Red;
```

При отображении фигур на поверхности оперируют такими понятиями как линия, за отображение которой отвечает класс *pen*, и заливка класс *Brush*.

Конструктор карандаша принимает два параметра цвет и толщину линии в пикселях.

```
Pen p1 = new Pen(c, 2);
```

Из свойств карандаша нужно отметить цвет и стиль линии `DashStyle`, задающее стиль отображения линии.

Класс *Brush* абстрактный, поэтому непосредственно экземпляр этого класса создать нельзя.

Можно создать экземпляр класса наследника, например, *SolidBrush* (сплошная заливка).

```
SolidBrush brush =new SolidBrush(Color.Red);
```

Разберём простой пример, отображения графики в контейнере, размещенном на форме. В данном случае в качестве контейнера выступает компонент панель. Пояснения ко всем строкам примера были приведены в тексте выше.

```
private void ButtonDraw_Click(object sender, EventArgs
e)
{
Graphics g = this.panel1.CreateGraphics();
g.Clear(Color.Red);
g.DrawLine(new Pen(Color.Green, 2),10,10,100,100);
g.DrawRectangle(new Pen(Color.DarkBlue, 3), 10, 10, 100,
100);
g.DrawEllipse(new Pen(Color.Black, 1), 15, 15, 125, 125)
}
```

Результат работы приведённого фрагмента кода представлен на рисунке 21.

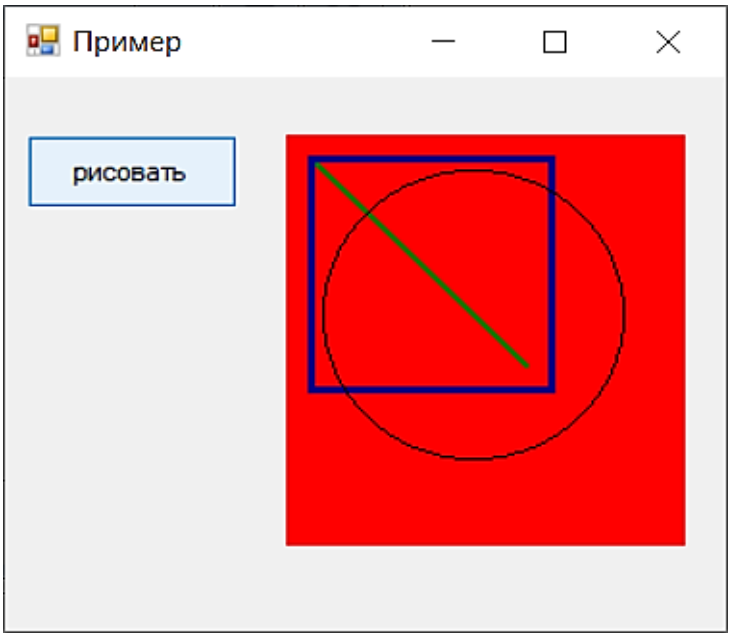



Рис. 21. Результат работы кода обработчика события нажатия на кнопку

Для работы с существующими в файлах изображениями используют объекты класса *Image*.

***Основные свойства класса Image***

`Height` – высота картинки;

`PixelFormat` – формат пикселя;

`RawFormat` – формат файла изображения;

`Size` – размер картинки в пикселях;

`Width` – ширина картинки.

***Основные методы класса Image***

`Clone( )` – вернуть копию объекта;

`GetBounds( )` – вернуть размеры картинки;

`GetThumbnailImage( )` – вернуть небольшое изображение;

`RotateFlip( )` – поворот изображения;

`Save( )` – сохранить в файл.

Приведенные в следующих строках код позволяет создать объект класса `Image` и загрузить в него изображения из файла.

```
Image image;
image =Image.FromFile (openDialog.FileName);
```

Метод `FromFile()` создаёт новый экземпляр класса `Image`, загружает в него картинку и возвращает объект в точку вызова.

Важно отметить, что загрузку картинки из файла рекомендуется делать в блоке исключений `try`, чтобы отследить возможную ошибку при загрузке картинки из файла, и корректно продолжить работу программы. Ошибка `OutOfMemoryException` может возникнуть в том числе и при некорректном формате файла.

Далее приведен код, позволяющий загрузить изображение из файла, отобразить его на панели, а также результат работы этого фрагмента кода и пояснения к его работе.

```
    private void ButtonFromFile_Click(object sender,
EventArgs e)
    {Image image;
    if ((openFileDialog1.ShowDialog() != Di-
alogResult.OK)) return;
    try
    {image = Image.FromFile(openFileDialog1.FileName);
    }
    catch (OutOfMemoryException ex)
    {
    MessageBox.Show("error");
    return;rom
    }
    Graphics g = this.panel1.CreateGraphics();
    g.DrawImage(image, 0,0, panel1.Size.Width,
panel1.Size.Height);
    }
```

Результат работы кода, приведённого в примере, показан на рисунке 22.

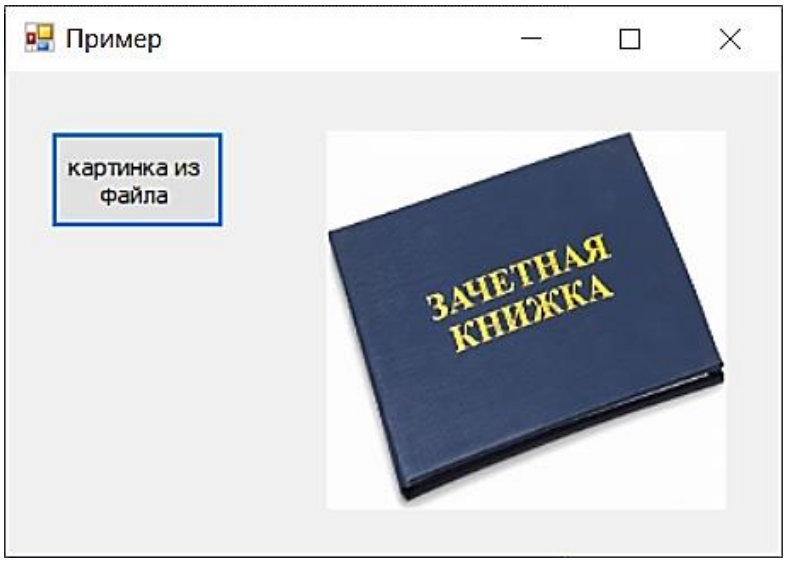


Рис. 22. Загрузка картинки из файла. Результат

В приведённом выше коде, создается объект класса `Image`. Далее идёт работа с добавленным на форму во время проектирования невизуальным компонентом `openFileDialog1`. Объекты класса `OpenFileDialog` можно создать и вовремя выполнения программы. Например.

```
OpenFileDialog openDialog = new OpenFileDialog();
openDialog.Filter = "изображения|*.jpg";
```

В первой строчке создаётся объект `openDialog` класса `OpenFileDialog.` Во второй установлено значение его свойства `Filter`. В первом примере значение свойства `Filter` компонента `openFileDialog1` установлено на шаге проектирования.

При не выбранном имени файла работа обработчика прерывается. Если имя файла выбрано, работа продолжается и при удачном открытии файла изображение передаётся в объект `image`.

Далее создаётся объект класса `Graphics`. Для созданного объекта вызывается метод `DrawImage`, отображающий объект `image` в заданных координатах. В нашем случае это координаты левого верхнего и правого нижнего угла. Т.е. картинка займёт всю поверхность панели.

## *Лабораторная работа № 8. Отображение простейших графических объектов*

***Цель работы***. Знакомство с принципами отображения графики в C#.

***Постановка задачи***. Разработать визуальное приложение, позволяющее отображать в окне, предложенные пользователю на выбор графические объекты.

Для этого на форме разместить компоненты, позволяющие организовать выбор одного из как минимум трёх видов рисунков (например, звезда, снеговик, дом), цвет, толщину линии, возможно задание других параметров.

Добавить на форму две кнопки, при нажатии на одну, объект отображается, на другую удаляется.

***Контрольные вопросы***

1. Назовите основной класс для работы с графикой в C#
2. В каком пространстве имён находится этот класс?
3. Назовите основные свойства этого класса.
4. Назовите основные методы этого класса.
5. Каким образом возможно программно задать цвет отображающейся фигуры?
6. Каким образом можно задать программно стиль отображения линии?
7. Каким образом можно задать координаты отображаемой фигуры?

***Задания для самостоятельной работы***

1) Создать приложение со следующим функционалом. При запуске программы, на форме отображён круг. При нажатии на кнопку круг передвигается вправо на один шаг, размер шага на усмотрение пользователя в пределах одного сантиметра.

2) К предыдущей работе добавить настройку цвета отображения фигуры.

3) Предыдущее задание расширить следующим функционалом: масштаб фигуры при перемещении должен изменятся (увеличиваться до определённого размера, затем уменьшаться до заданного значения).

4) К предыдущей работе добавить регулятор скорости перемещения фигуры по траектории.

5) К предыдущей работе добавить регуляторы диапазона и скорости масштабирования.

6) Дополнить предыдущее задание следующим образом, добавить вращение и регулировку скорости вращения фигуры.

7) Разработать приложение: «Графический редактор», обладающий перечисленным далее функционалом. Возможность выбора инструмента (минимальный набор: линия, карандаш, прямоугольник, эллипс), возможность очистки и заливки фона определённым цветом, выбор цвета и толщины линии инструмента. Расширить функционал на усмотрение разработчика.

8) Создать приложение, при запуске которого в появившемся окне отображаются две фигуры: круг и квадрат. Организовать с помощью обработчиков соответствующих событий перемещение фигур по поверхности формы. Начало движения – нажатие кнопки мыши на поверхности фигуры. Движение – движение мыши, при удержании её кнопки нажатой (если кнопка была нажата на фигуре). Завершение движения – кнопка мыши опушена. Использовать следующие события *MouseDown, MouseMove, MouseUp,* класса *Form.* Ограничение: фигуры не должны пересекать границы клиентской области формы. Предусмотреть класс фигура (*Figure*) и его потомков: круг (*Circle)*, квадрат (*Square*). Метод перемещения фигур должен быть полиморфным.

9) Создать приложение, при запуске которого на форме отображается линия. При наведении курсора на линию изменяются её толщина и цвет линии. Нажатие в этом состоянии на левую клавишу мыши изменяет форму курсора. Если курсор находился на конце линии, то форма курсора отражает состояние «резиновая нить», в этом состоянии можно изменить положение конца линии путем перемещения мыши в другую точку. Если курсор находился в средней части линии, то форма курсора отражает состояние «перенос линии», в этом состоянии перемещение мыши приводит к параллельному перемещению линии по экранной форме. Отпускание мыши приводит задачу в исходное состояние, при этом линия остается в состоянии, которое она приняла в результате манипуляций. Для реализации необходимо использовать события класса *Form: MouseDown, MouseMove, MouseUp.* Требование: линия должна моделироваться отдельным классом.

10) Разработать визуальное приложение, при запуске программы на форме приложения появляется геометрическая фигура, которая будет рассмотрена как траектория движения другой фигуры. Предусмотреть: регулятор изменения размера (масштабирования) траектории, настройку цвета фона рабочей области, цвета прорисовки траектории. Центр траектории всегда должен совпадать с центром клиентской области формы. Траектория не должна выходить за пределы рабочей области. При уменьшении линейных размеров траектории масштаб не должен становиться отрицательным (траектория не должна «выворачиваться наизнанку»). Для моделирования траектории разработать соответствующий класс.

## Организация меню в приложении

Компонент ***MenuStrip*** позволяет организовать главное меню. При его добавлении на форму, под строкой заголовка размещается панель меню. При этом, имя меню, по умолчанию это *menuStrip1*, становится значением свойства *MainMenuStrip* формы.

Добавление разделов меню и создание подпунктов меню не вызывает затруднений, для этого нужно щелкнуть на прямоугольнике с надписью: «Вводить здесь» и ввести нужный текст. При создании подпунктов меню, предлагается выбрать тип, это может быть элемент меню или выпадающий список, поле для ввода, или разделитель.

Двойной щелчок по пункту меню создаёт обработчик события *Click*.

На рисунке 23 показан процесс создания пунктов и подпунктов меню.


Рис. 23. создание пунктов и подпунктов меню

Дале приведён код обработчика события нажатия на пункт меню «Закрыть». Код состоит из вызова одного метода `Close()`, работа которого приводит к закрытию формы.

```
private void ЗакрытьToolStripMenuItem_Click(object sender, EventArgs e) {Close();}
```

Компонент ***ContextMenuStrip***, рисунок 24 позволяет создать контекстное меню для компонентов. Контекстное меню появляется при щелчке правой кнопкой мыши на элементе управления или форме.

*ContextMenuStrip* это невизуальный компонент. Такое меню может содержать только подпункты, разделы добавлять нельзя.

После создания нужных подпунктов, необходимо, связать меню с тем компонентом, для которого оно создавалось. Для этого выбираем нужный элемент управления и в выпадающем списке свойства *ContextMenuStrip* выбираем нужное нам меню.


Рис. 24. Компонент *ContextMenuStrip*
а) задание пунктов меню; б) связь с компонентом *button1*

Компонент ***ToolStrip*** позволяет создать панель инструментов. По умолчанию при добавлении компонент расположен по правому краю формы. На панели предлагается разместить компоненты следующих видов: кнопка, метка, кнопка с всплывающим меню, разделитель, текстовое поле, индикатор прогресса, выпадающий список. На рисунке 25 представлено открытое окно добавления инструментов.

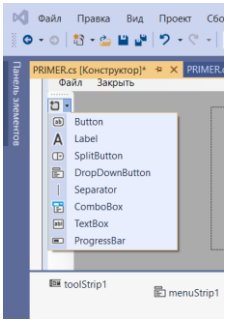



Рис. 25. Добавление инструментов на панель

Компонент ***StatusStrip*** предназначен для отображения состояния работы программы. На поверхности этого компонента можно размещать другие элементы, которые будут нести информацию о состоянии работы.

Процесс выбора таких элементов представлен на рисунке 26.

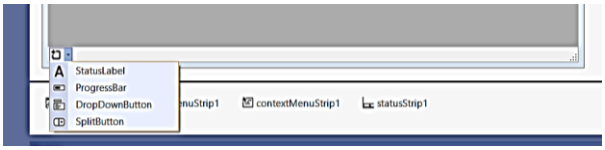



Рис. 26. Добавление элементов в компонент StatusStrip

## Диалоги

Библиотека *.NET Framework* имеет два типа компонентов: визуальные и невизуальные. Об этом было упомянуто ранее. Напомним. Визуальные компоненты являются элементами пользовательского интерфейса. Это, например, компоненты: кнопка (*Button*), выпадающий список (*comboBox*) или метка (*Label*).

Невизуальные компоненты не имеют пользовательского интерфейса и не могут располагаться на форме во время проектирования. Дизайнер *Visual Studio* располагает их внизу окна дизайнера. Такими компонентами являются, например, компоненты работы с базами данных, таймер (*Timer*), контекстное меню, различные диалоговые окна.

Компонент ***SaveFileDialog*** расположенный на вкладке *Dialogs* панели *Toolbox*, позволяет сохранить информацию в файле с выбранным именем.

Основные свойства этого компонента.

*InitialDirectory* - возвращает или задает начальную папку, отображенную диалоговым окном файла. (Унаследовано от *FileDialog*)

*FileName* - возвращает или задает строку, содержащую имя файла, выбранного в диалоговом окне. (Унаследовано от *FileDialog*)

*Filter* - возвращает или задает текущую строку фильтра имен файлов, которая задает варианты, появляющееся в окне "сохранить как тип файла" или "файлы типа" в диалоговом окне. (Унаследовано от *FileDialog*)

*FilterIndex* - возвращает или задает индекс фильтра, выбранного в настоящий момент в диалоговом окне файла. (Унаследовано от *FileDialog*)

При необходимости изменить заголовок рассматриваемого диалогового окна, используют свойство *Title*.

*OpenFile* - открывает выбранный пользователем файл с разрешением на чтение и запись.

*LoadFile(string)* - загружает файл в формате *RTF* или стандартный текстовый файл в кодировке *ASCII* в элемент управления *RichTextBox*.

Для того что бы сохранить текст, например, из компонента *richTextBox* в файл необходимо в обработчике соответствующего события (нажатия на кнопку или щелчок по соответствующему пункту меню) активизировать Диалоговое окно компонента *SaveFileDialog*, задать необходимое имя и тип файла

Делается это следующим образом.

```
    saveFileDialog1.Filter = "txt (*.txt)|*.txt|rtf
(*.rtf)|*.rtf|All files|*.*";
    if ((saveFileDialog1.ShowDialog() == Di-
alogResult.OK)
    &&saveFile1.FileName.Length > 0)
                {string fileName = saveFileDia-
log1.FileName;
                    richTextBox1.SaveFile(fileName);
                }
```

Ниже приведён фрагмент кода, отображающий в компоненте *richTextBox* текст из выбранного файла

```
    if ((openFileDialog1.ShowDialog() == Di-
alogResult.OK) && (saveFileDialog1.FileName.Length)>0)
{
string fileName = openFileDialog1.FileName;
richTextBox1.LoadFile(fileName);
}
```

## *Лабораторная работа № 9. Работа с файлами, диалоговыми окнами, создание главного и контекстного меню. Приложение «Анкета»*

***Цель работы***

Закрепление навыка написания обработчиков событий.

Знакомство с принципами организации в приложении главного и контекстного меню, продолжение изучения компонент отображения текстовой информации.

Организация хранения информации в файлах и работы с данными, хранимыми в файле.

***Постановка задачи***.

Разработать приложение «Анкета», позволяющее организовать получении информации от пользователя о каком-либо объекте, через заполнение им соответствующих полей. Организовать возможность сохранения этой информации в файле, возможность чтения информации из файла и отображения в соответствующем окне формы.

***Указания к выполнению работы.***

На форме разместить два компонента *panel,* разделив их компонентом *splitter*.

Первую панель выровнять по левому краю, вторую по ширине.

На левой панели, с целью сбора информации об объекте, разместить компоненты отображения текстовой информации: *TextBox, CheckedListBox, ComboBox, ListBox, CheckBox, DateTimePicker* и др.

На правой панели разместить компонент, в котором будет отображена вся введенная в левом окне информация, например, компонент *richTextBox*. Перенос информации осуществить в обработчике события *Click* компонента *Button*, расположенного на правой панели.

Панели и все размещенные на них объекты, должны сопровождаться пояснительными метками. Для панелей и меток изменить значения по умолчанию свойств *Color* и *Font* и его подсвойств *Size, Style, Color*.

На правой панели разместить три кнопки. Кнопку, нажатие на которую приводит к копированию информации из левой панели в окно *richTextBox*. Кнопку, активирующую диалоговое окно, позволяющее сохранить всю информацию из правой панели в текстовый файл. И кнопку, нажатие на которую позволяет выбрать файл информация из которого будет загружена в окно *richTextBox*.

Эти действия должны быть продублированы пунктами главного меню.

Компонент *richTextBox* должен иметь контекстное меню, содержащее пункты: открыть, сохранить, очистить. По желанию меню можно добавить пункты меню.

***Вопросы для самоконтроля***

1. Перечислите известные Вам компоненты контейнеры.
2. Какие компоненты лучше использовать для отображения заголовков контейнеров?
3. Какие компоненты используют для организации выбора пользователям одного из двух вариантов?
4. Используя какие компоненты можно организовать выбор пользователем нескольких вариантов их списка?
5. Используя какие компоненты можно организовать выбор одного варианта из нескольких предложенных?
6. Какой компонент позволяет организовать выбор пользователем даты и времени, какие свойства этого компонента для этого нужно использовать?
7. Какой компонент используют для создания главного меню?
8. Каким образом можно добавить новый пункт в главное меню?
9. Что такое контекстное меню?
10. Каким образом можно создать контекстное меню для конкретного компонента?
11. Каким образом возможно организовать подсказки, всплывающие при наведении курсора на объект?
12. Какие компоненты называют визуальными?
13. Какие компоненты называют невизуальными?
14. Каким образом программист может предоставить пользователю возможность сохранить информацию в файле?
15. Каким образом программист может предоставить пользователю возможность прочитать информацию из файла?

***Задания для самостоятельной работы***

Во всех заданиях необходимо разработать визуальное приложение на языке *C#*. Организовать ввод пользователем всей необходимой информации, предусмотреть ситуации ошибок пользовательского ввода, отображение всей введённой информации в отдельном окне, с

возможностью его редактирования, очистки, записи данных в файл и отображение в окне информации из выбранного файла.

1) Разработайте приложение «Домашняя библиотека», позволяющее пользователю просмотреть все имеющиеся книги и добавить описание новой.

2) Разработайте приложение «Картинная галерея», позволяющее пользователю просмотреть информацию обо всех имеющихся картинах и добавлять описание новых.

3) Разработайте приложение «Зоопарк», позволяющее пользователю просмотреть информацию о всех животных и добавить описание новых.

4) Разработайте приложение «Ветеринарная клиника», позволяющее пользователю просмотреть информацию о всех пациентах, и добавить новых.

5) Разработайте приложение «Магазин игрушек», позволяющее пользователю просмотреть информацию о существующих товарах и добавить новые.

6) Разработайте приложение «Кафе», позволяющее пользователю просмотреть информацию о существующих блюдах и добавить новые.

7) Разработайте приложение «Каталог косметики», позволяющее пользователю просмотреть информацию о существующих товарах и добавить новые.

8) Разработайте приложение «Красная книга Владимирской области», позволяющее пользователю просмотреть информацию о редких животных области и добавить новые виды.

9) Разработайте приложение «Гербарий», позволяющее пользователю просмотреть информацию о существующих экспонатах и добавить новые.

10) Разработайте приложение «Филателист», позволяющее пользователю просмотреть информацию о существующих марках и их количестве в коллекции и добавить новые.

## ПРИМЕРЫ ИТОГОВЫХ ЗАДАНИЙ

Все идентификаторы, использованные в коде должны информативными. Т.е. из имени должен быть понятен тип и назначение объекта. Не должно быть имён типа *mas, ob1, label1, storona* и т.п.

***Варианты заданий***

1) Разработать визуальное приложение на языке C#.

Описать классы Квадрат и Прямоугольник, наследник класса Квадрат.

Класс Прямоугольник должен содержать:

- соответствующие поля и свойства для хранения информации о его сторонах;

- методы вычисления периметра и площади (методы должны возвращать значения, а не выводить их);

- описать конструктор с параметрами (два параметра - стороны прямоугольника).

Объявить массив из пяти объектов этого класса.

На форме разместить:

- компоненты для ввода пользователем значений сторон, указать единицы измерения;

- кнопку с надписью: «Создать», при нажатии на которую создаётся и записывается в массив объект класса прямоугольник, предусмотреть проверку корректности заполнения полей;

- кнопку – «Площадь», в обработчике события нажатия на которую вычисляется и отображается на форме сумма площадей всех созданных прямоугольников (при отображении не забыть указать единицы измерения).

При добавлении объекта, на форме отобразить значение его сторон, периметра и площади (при отображении не забыть указать единицы измерения) При добавлении нового объекта информация о предыдущих остаётся видна.

По нажатию на кнопку «Создать», поочерёдно добавить в массив пять объектов. После добавления пятого последнего объекта, кнопку с надписью: «Создать» сделать не доступной, изменив, для этого значение соответствующего свойства, кнопку с надписью: «Площадь» сделать видимой. При запуске приложения эта кнопка должна быть невидима.

2) Разработать визуальное приложение на языке C#.

Описать классы Квадрат и Прямоугольный параллелепипед, наследник класса квадрат.

Класс Прямоугольный параллелепипед должен содержать:

- соответствующие поля и свойства для хранения информации о его сторонах;

- методы вычисления площади поверхности и объёма (методы должны возвращать значения, а не отображать их на форме);

- описать конструктор с параметрами (два параметра - стороны прямоугольника).

Объявить массив из пяти объектов этого класса.

На форме разместить:

- компоненты для ввода пользователем значений сторон, указать единицы измерения;

- кнопку с надписью: «Создать», при нажатии на которую создаётся объект класса Прямоугольный параллелепипед, созданный объект добавляется в массив, предусмотреть проверку корректности заполнения полей;

- кнопку «Площадь», в обработчике события нажатия на которую вычисляется и отображается на форме сумма площадей всех созданных прямоугольников (при отображении не забыть указать единицы измерения). Кнопка невидима до создания всех объектов.

При добавлении объекта, на форме отобразить значение его сторон, площади поверхности и объёма (при отображении не забыть указать единицы измерения).

По нажатию на кнопку «Создать» поочерёдно добавить в массив пять объектов класса наследника. После добавления пятого последнего объекта, кнопку «Создать» сделать не доступной, изменив, для этого значение соответствующего свойства, и сделать видимой кнопку «Площадь», невидимую при начале работы приложения.

3) Разработать визуальное приложение на языке C#.

Описать базовый класс Точка и класс наследник Круг.

Класс наследник должен содержать:

- поле и свойство для хранения значения радиуса;

- методы вычисления длины дуги окружности и площади (методы должны возвращать значения, а не отображать их на форме);

- конструктор с параметрами.

Объявить массив из пяти объектов этого класса.

На форме разместить:

- компоненты для ввода пользователем значения радиуса, указать единицы измерения;

- кнопку «Создать», при нажатии на которую создаётся и записывается в массив объект класса круг, предусмотреть проверку корректности введённого значения;

- кнопку, в обработчике события нажатия на которую вычисляется и отображается на форме сумма площадей всех созданных объектов класса круг (при отображении не забыть указать единицы измерения).

При добавлении объекта, на форме отобразить значение радиуса, длины дуги окружности и площади (при отображении не забыть указать единицы измерения).

Добавить в массив пять объектов. Добавлять по одному элементу по нажатию на кнопку «Создать».

После добавления пятого последнего объекта, кнопку «Создать» сделать не доступной, изменив, для этого значение соответствующего свойства, и сделать видимой кнопку «Площадь», нажатие на которую приводит к вычислению и отображению на форме суммы площадей всех объектов массива, написать соответствующий обработчик.

Разработать визуальное приложение на языке C#.

Описать базовый класс круг и класс наследник - Сфера.

Класс наследник должен содержать:

- поле и свойство для хранения информации о радиусе;

- методы вычисления площади поверхности сферы, объема сферы (методы должны возвращать значения, а не отображать их);

- конструктор с параметрами.

Объявить массив из пяти объектов этого класса.

На форме разместить:

- компоненты для ввода пользователем значения радиуса, указать единицы измерения;

- кнопку с надписью: «Создать», при нажатии на которую создаётся и записывается в массив объект класса Сфера, предусмотреть проверку корректности введённого значения;

- кнопку с надписью: «Площадь», в обработчике события нажатия на которую вычисляется и отображается на форме сумма площадей

всех созданных объектов класса Сфера (при отображении не забыть указать единицы измерения).

При добавлении объекта, на форме отобразить значение радиуса, площади поверхность и объёма сферы (при отображении не забыть указать единицы измерения).

После добавления пятого последнего объекта, кнопку «Создать» сделать не доступной, изменив, для этого значение соответствующего свойства. Кнопку «Площадь» сделать видимой. При запуске приложения кнопка «Площадь» невидима.

# ЗАКЛЮЧЕНИЕ

Работа по подготовке специалистов в области математики и компьютерных наук предполагает освоение ими дисциплин программирования и получении навыков в области разработки, отладки, тестирования и модификации программного обеспечения. Работа специалиста в этой области немыслима без перечисленных выше навыков.

Освоение курса «Объектно-ориентированное программирование» даёт студенту возможность получения этих навыков в части разработки классов, использования ранее написанного кода, в том числе и другими разработчиками, создания визуальных приложений.

Полученные знания и навыки будут применены при решении широкого спектра учебных и профессиональных задач.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Павловская Т. А., C#. Программирование на языке высокого уровня. Учебник для вузов. — СПб.: Питер, 2009. — 432 с: ил. ISBN 978-5-91180-174-8

2. Павловская Т. А., Щупак Ю. А. П12 C++. Объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2006. — 265 с: ил. ISBN 5-94723-842-X

3. Страуструп, Б. Программирование. Принципы и практика с использованием C++ / Б. Страуструп. – М. : Вильямс, 2016. – 1328 с. – ISBN 978-5-8459-1949-6. 5.

4. Шилдт. Г. Полный справочник по C++ : пер. с англ. / Г. Шилдт. – 4-е изд., стер. – М. : Вильямс, 2015. – 800 с. – ISBN 978-5-8459-2047-8.

5. Троелсен Э., Джепикс Ф., Pro C# 7: With .NET and .NET, Core/ Троелсен Э., Джепикс Ф.,. М. : Вильямс, 2018. – 1328 с. ISBN 978-5-6040723-1-8, 978-1-4842-3017-6

6. Фленов М., Библия C# 4-е изд., переработанное и дополненное. - СПб.: БХВ-Петербург, 2020, 512 с. ISBN 978-5-9775-4041-4

# ПРИЛОЖЕНИЯ

***Приложение 1***

## Образец титульного листа отчёта по лабораторной работе

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Владимирский государственный университет имени Александра Григорьевича и Николая Григорьевича Столетовых»
(ВлГУ)

Кафедра ФиПМ

**ЛАБОРАТОРНАЯ РАБОТА №4**

по дисциплине «Объектно-ориентированное программирование»
на тему: «Шаблоны классов»

Выполнил:
ст. группы МКН-119
Иванов И.И.
Принял:
ст. преподаватель
каф. ФиПМ
Шишкина М. В.

Владимир 2020

***Приложение 2***

## Требования к форматированию отчёта по лабораторной работе

Отчёт по лабораторной работе должен быть создан в текстовом процессоре Microsoft Word. Отчёт должен быть распечатан на одной стороне чистого белого листа формата А4, не допускается использование оборотной стороны использованных листов.

Титульный лист должен быть оформлен в соответствии с примером, приведенным в приложении 1.

При наборе текста шрифт *Times New Roman*, кегль 14 пт.

Поля задать следующим образом: левое поле – 30 мм; правое – 10 мм; верхнее и нижнее – 20 мм. междустрочный интервал – полуторный; режим выравнивание «по ширине»; отступ в начале абзаца (красная строка) – 15 мм.

Все листы должны быть пронумерованы. Титульному листу номер присваивают, но не проставляют. Следовательно, номер страницы, следующий после титульного листа – 2. Номер проставляют по центру листа, кегль 12 пт.

Для допуска к защите лабораторной работы необходимо продемонстрировать работоспособность написанной программы, представить отчёт написанный и оформленный в соответствии с требованиями.

Для защиты работы необходимо ответить на вопросы по теоретической части работы, подробно пояснить строки кода программы, указанные преподавателем. Самостоятельно выполнить задание, выданное преподавателем по теме и уровню сложности соответствующее заданию лабораторной работы

*Приложение 3*

## Требования к содержанию отчёта

Отчёт по лабораторной работе должен содержать следующие части.

1. Титульный лист, оформленный по образцу, приведённому в приложении 1.
2. Цель работы.
3. Постановку задачи.

Формулировка цели работы и постановки задачи должны точно соответствовать, указанным преподавателем.

4. Пять – семь ключевых слов на русском и английском языках. В качестве ключевых слов стоит выбирать слова опираясь на которые можно составить исчерпывающий ответ по теме работы.
5. Краткая теоретическая часть должна содержать от двух до четырёх страниц связного самостоятельно написанного текста с примерами. Теоретическая часть не должна быть копией лекций, но Лекционный материал должен быть взят за основу. При этом должны быть рассмотрены все вопросы, затронутые в работе, с теоретической точки зрения и на примерах, реализованных автором.
6. Ход работы должен содержать программный код, написанный на изучаемом языке программирования и несколько тестов, результатов работы программы с объяснением полученных результатов.
7. Выводы по работе. В этой части необходимо указать к какому выводу пришёл автор во время выполнения работы. Например, о достоинствах и недостатках изученного метода, разработанного алгоритма и т.п., для решения какого класса задач используют этот подход.

***Приложение 4***

## Типы данных языка C#

| Типы значений *(хранят само значение)* | Ссылочные типы *(хранят ссылку на значение)* |
|---|---|
| - целочисленные типы | - строки |
| - вещественные | - объекты |
| - символы | |
| - логический | |

***Диапазон целочисленных переменных в C#***

| Тип | Разрядность биты/байты | Диапазон |
|---|---|---|
| byte | 8/1 | 0:255 |
| sbyte | 8/1 | -128:127 |
| short | 16/2 | -327 68 : 327 67 |
| ushort | 16/2 | 0 : 655 35 |
| int | 32/4 | -217483648 : 2147483647 |
| uint | 32/4 | 0 : 4294967295 |
| long | 64/8 | -9223372036854775808 : 9223372036854775807 |
| ulong | 64/8 | 0 : 18446744073709551615 |

Несмотря на то, что тип `char` определен в *C#* как целочисленный, в *C#* отсутствует автоматическое преобразование символьных значений в целочисленные и обратно.

В C# не определено взаимное преобразование логических и целых значений. Логический тип имеет только два значения *true* и `false.`

***Приложение 5***

## Преобразование типов в языке C#

Т.к. C# строго типизированный язык, выполнение операций возможно только на однотипных переменных.

При использовании в одном выражении разнотипных переменных прибегают к приведению типов.

***Неявное приведение*** типов происходит автоматически, при выполнении двух условий: типы совместимы и диапазон представления значений типа переменной приёмника шире, чем у типа переменной источника.

```
short s=7;
int i=s;
```

Обратное не верно. Для этого необходимо выполнить ***явное приведение*** типа.

```
int i1 = 128;
Console.WriteLine((short)i1);
```

Чтобы выполнить преобразование между несовместимыми типами используют методы класса *Convert* из пространства имен *System* и методы *Parse* и *TryParse* встроенных числовых типов, например, *Int32.Parse*.

Для преобразования строки в число следует использовать методы ***Parse*** или ***TryParse.*** Метод *Parse* следует использовать если у программиста есть уверенность, что такое преобразование возможно.

```
byte d = byte.Parse("4");
```

Для обработки данных введённых пользователем предпочтительнее использовать метод `TryParse`

```
public static bool TryParse (string s, out double result);
```

где *s* это строка, содержащая преобразуемое число и `result` числового типа, конкретный тип зависит от того для какого класса вызван метод.

```
if (Double.TryParse(value, out number))
Console.WriteLine("'{0}' --> {1}", value, number);
Пример использованияметодов класса Convert.
int var1 = 8, var2 = 5;
byte sum = Convert.ToByte(var1+var2);
```

***Приложение 6***

## Работа с массивами в языке C#

Массив представляет собой совокупность элементов одного типа, объединённых под одним именем.

Индексация элементов массива в языке C# так же, как и в языке *C++* начинается с нуля. Синтаксис объявления массива.

```
<тип данных элементов массива> [ ] <имя массива>;
int [] mas;
```

При создании экземпляра массива, используя оператор ***new.***

```
<имя массива>= new <тип данных элементов> [<количество элементов>];
mas = new int[3];
```

Эти два шага можно объединить в одной строке.

```
<тип данных элементов массива> [ ] <имя массива>= new <тип данных элементов> [<количество элементов>];
int[] mas1 = new int[3];
```

Установить значения элементов массива можно тремя способами.

- вручную, обращаясь к конкретному элементу;
- при циклическом переборе элементов;
- при объявлении массива.

```
foreach (int i in myArr) Console.WriteLine(i);
```

У каждого массива имеется свойство, хранящее количество элементов этого массива. Это свойство *Length*. Свойство только для чтения. Это свойство удобно использовать при циклическом переборе элементов одномерного массива.

Для этого необходимо указать имя свойства, после имени массива, разделив их точкой с запятой.

<имя массива>.<имя свойства>

```
mas1.Length
```

При запросе длины многомерного массива, возвращается общее число элементов, из которых может состоять массив.

```
for (int i=0; i<mas1.Length; i++ ) {mas1[i]= i*2; }

int[,] myArr = new int[4, 5];
```

```
Random ran = new Random();
```

Пример инициализации массива случайными числами от 1 до 15.

```
for (int i = 0; i < 4; i++)
{for (int j = 0; j < 5; j++)
{myArr[i, j] = ran.Next(1, 15); }
}
```

При необходимости перебрать элементы массива, как и любой другой коллекции используют цикл позволяет перебирать элементы коллекции (объект, представляющий список других объектов, например, массив).

В *C#* определено несколько видов коллекций, каждая из которых является массивом. Ниже приведена общая форма оператора цикла

```
foreach (<тип имя переменной цикла> in <коллекция>)
<оператор;>
```

```
foreach (int item in mas) Console.WriteLine("elem
={0}", item);
```

В основе каждого типа в системе типов *.NET* (в том числе фундаментальных типов данных) в конечном итоге лежит базовый класс `System.Object`.

*Учебное электронное издание*

ШИШКИНА Мария Викторовна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Практикум

*Издается в авторской редакции*

***Системные требования:*** Intel от 1,3 ГГц ; Windows XP/7/8/10; Adobe Reader; дисковод CD-ROM.

**Тираж 26 экз.**

Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых
Изд-во ВлГУ
rio.vlgu@yandex.ru

Институт прикладной математики, физики и информатики
кафедра физики и прикладной математики
msh@vlsu.ru