Владимирский государственный университет

М. В. ШИШКИНА

ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Практикум

Владимир 2025

Министерство науки и высшего образования Российской Федерации
Федеральное государственное образовательное учреждение
высшего образования
Владимирский государственный университет имени
Александра Григорьевича и Николая Григорьевича Столетовых

М. В. ШИШКИНА

ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Практикум

Электронное издание

Владимир 2025

ISBN 978-5-9984-2025-2

УДК 004.42
ББК 32.973.26-018

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

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

Издается по решению редакционно-издательского совета ВлГУ

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, 2025. – 116 с. – ISBN 978-5-9984-2025-2. – Электрон. дан. (3,56 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.

Изложены теоретические основы объектно ориентированного подхода, разобрано достаточное количество примеров, приведены задания по лабораторным работам, задания для самостоятельной работы и контрольные вопросы для самоконтроля и закрепления материала. 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 «Объектно ориентированное программирование».

Предназначен для студентов бакалавриата направлений подготовки 12.03.05 – Лазерная техника и лазерные технологии, 01.03.02 – Прикладная математика и информатика, 02.03.02 – Фундаментальная информатика и информационные технологии.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Ил. 26. Табл. 1. Библиогр.: 6 назв.

ISBN 978-5-9984-2025-2

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Практикум позволяет обучающимся овладеть навыками разработки объектно ориентированного подхода в программировании от описания первых классов до разработки визуальных приложений с организацией диалога с пользователем. Издание содержит достаточное количество теоретических сведений и примеров для выполнения заданий лабораторных работ, самостоятельной работы, других материалов, входящих в состав учебно-методического комплекса дисциплины «Объектно ориентированное программирование», разработанного по современным федеральным государственным образовательным стандартам подготовки бакалавров по направлению 12.03.05 «Лазерная техника и лазерные технологии».

Структура практикума организована таким образом, что каждая следующая работа основана на знаниях и навыках, полученных при выполнении предыдущей. Все лабораторные задания предваряются подробной теоретической информацией, содержащей достаточное количество примеров. После каждого задания для лабораторной работы следует перечень вопросов, помогающих студенту самостоятельно оценить уровень своих знаний и закрепить их. Каждая тема содержит задания для самостоятельной работы, позволяющие закрепить навыки, полученные во время выполнения лабораторной работы.

В прил. 1 – 3 приведены требования к отчету по работе и рекомендации по форматированию отчёта, в прил. 4 – 6 представлена краткая справочная информация по языкам программирования C++ и C#.

ВВЕДЕНИЕ

Быстрое развитие информационных технологий во всех областях современной жизни привело к увеличению затрат времени на разработку, сопровождение и модификацию программных продуктов. Ответом на возникшие сложности стало появление концепции объектно ориентированного программирования.

В настоящие время объектно ориентированный подход используется во многих языках программирования. Свободное владение навыками объектно ориентированного программирования открывает перед разработчиком мощный ресурс для создания востребованных в современном мире визуальных приложений. Возможность описать на языке программирования предметы бытовой и профессиональной сферы сокращает программный код, позволяет сделать его более наглядным и читаемым, уменьшая время разработки и модификации, упрощая процесс командной работы.

Практикум состоит из двух больших взаимосвязанных разделов. Первый раздел базируется на изученных студентами в предыдущих курсах основах программирования, знании основ языка программирования высокого уровня *C++*, принципов структурного и модульного программирования, основных алгоритмических структур, умении реализовывать на языке программирования высокого уровня комбинированные алгоритмы, представленные в виде словесного описания, математической формулы, блок-схемы.

Освоение материала, приведённого в первом разделе, даёт возможность обучающимся получить представление о парадигме объектно ориентированного программирования.

Второй раздел основан на знаниях и навыках, полученных в ходе работы с первым разделом, и позволяет студентам закрепить и углубить свои знания и навыки в области объектно ориентированного программирования, расширить представления о языках программирования высокого уровня, овладеть основами разработки визуальных приложений.

Раздел 1
ПРИНЦИПЫ ОБЪЕКТНО ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Введение в объектно ориентированное программирование

Теоретические сведения и методические указания к работе

Объектно ориентированное программирование возникло в результате поиска выхода из ситуации, когда сложность реальных производственных задач порождала большое количество подпрограмм, когда традиционное процедурное программирование уже не справлялось с задачами такой сложности. Декомпозиция, т. е. разбиение задачи на простые шаги в рамках структурного программирования, давала большое количество функций, что приводило к различным сложностям, в том числе из-за необходимости именования достаточно большого количества переменных. Работа нескольких программистов над проектом приводила к большим временным затратам на сопровождение программного продукта.

В объектно ориентированном подходе эта проблема решена следующим образом: предполагается принять в качестве критерия декомпозиции принадлежность элементов к различным абстракциям предметной области. Эти абстракции, в свою очередь, описываются на языке программирования и обладают состоянием и поведением. Таким образом, код, написанный с использованием парадигмы объектно ориентированного программирования, представляет собой набор описаний абстрактных сущностей, обладающих состоянием и поведением, способов взаимодействия этих сущностей. Во время исполнения программы реализуется один из возможных сценариев создания экземпляров, описанных сущностей и их взаимодействия.

Рассмотрим подробнее такой подход. Пользовательский тип, объединяющий в себе разнотипные данные и функции, обрабатывающие эти данные, называют *классом,* включённые в класс данные называют полями, или атрибутами класса. Функции, объявленные в классе, называют *методами* класса, или функциями-членами класса. Совокупность методов класса определяет поведение экземпляра соответствующего класса. Под поведением понимают способ взаимодействия между экземпляром класса (переменной соответствующего типа) и другими сущностями, реакции экземпляров на внешние воздействия класса.

Переменные типа класс (экземпляры соответствующего класса) называют *объектами* этого класса.

Для описания класса необходимо указать ключевое слово `class` и сразу за этим словом – имя класса, после чего в фигурных скобках указать все поля-данные и заголовки методов этого класса. Заканчивается описание класса точкой с запятой.

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

В качестве примера опишем класс `Kvadrat`, содержащий одно вещественное поле и один метод. Поле `float a` предназначено для хранения стороны квадрата, а метод `Sq()` – для вычисления площади квадрата.

Обычно описание класса производится в заголовочном файле и содержит лишь сигнатуру методов; описание этих методов вынесено в отдельный модуль с тем же именем, что и у заголовочного файла, и расширением *cpp*. Для наглядности и простоты изложения в этом и следующих примерах описание методов будет приведено сразу при описании класса. Этот подход следует использовать только при знакомстве с объектно ориентированным программированием и написании первых классов. В дальнейшем рекомендуется разделять объявление и описание классов.

```
class Kvadrat
{   float a;// поле – сторона квадрата;
public:
    float Sq() { return a*a; };
};
```

Одно из основных свойств объектно ориентированного программирования – *инкапсуляция*. Под инкапсуляцией понимают объединение данных, описывающих одну сущность, и методов для обработки этих данных. Данные объекта необходимо защитить от возможности изменения извне. Для ограничения доступа к членам класса используют *модификаторы доступа*.

Модификаторы доступа – это ключевые слова, использование которых позволяет сделать данные и методы доступными или закрытыми для внешнего использования, т. е. использования за пределами класса.

Всего в языке программирования *C++* существует три модификатора доступа: `private`, `private` и `protected`. Для применения модификатора к членам класса необходимо написать соответствующее ключевое слово и поставить двоеточие. Действие модификатора распространяется с момента указания ключевого слова на все объявленные ниже члены класса до тех пор, пока не будет встречен другой модификатор доступа или не закончится описание класса. Рассмотрим влияние каждого из этих модификаторов на возможности работы с членами класса.

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

Инкапсуляция предполагает сокрытие данных от внешних изменений, поэтому принято поля-данные задавать с модификатором `private`. Методы, предназначенные для вызова из внешних функций, задают с модификатором `public`; метод, объявленный с модификатором `private`, нельзя вызвать за пределами класса.

При реализации сценария взаимодействия с объектом класса может возникнуть необходимость получения или изменения значения полей-данных объекта. Для достижения этой цели без нарушения принципа инкапсуляции используют `Get-` и `Set`-методы, о которых будет рассказано позднее.

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

По умолчанию к членам класса применяется модификатор доступа `private`. Поэтому 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, так как при его создании вызывается специфический метод класса, называемый *конструктором*.

Конструкторы

Конструктор – это специальный метод класса, вызываемый каждый раз при создании объектов этого класса и инициализирующий его поля.

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

Так как конструктор – это метод, а метод – это функция, его объявление и описание происходит по тем же правилам, что и для любых других функций, с одним исключением: в заголовочной строке не указывается тип возвращаемого значения, даже тип `void`.

Конструктор для класса, приведенного выше в качестве примера, может быть описан так:

```
Kvadrat() { … };
```

или так:

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

Конструктор как любая пользовательская функция может быть перегружен необходимое число раз. Таким образом можно создать необходимое количество конструкторов для разных способов инициализации объектов. При этом важно избегать возможных конфликтов при вызове перегруженных функций, т. е. при вызове конструктора должно быть однозначно определено, какой из перегруженных вариантов должен быть вызван. Понятно, что все конструкторы класса будут иметь одинаковое имя – имя класса – и разный список формальных параметров.

Все конструкторы можно отнести к одному из четырех видов:

– *конструктор по умолчанию;*

– *конструктор с параметрами;*

– *конструктор с параметрами по умолчанию;*

– *конструктор копирования.*

Если конструктор не имеет параметров, то он носит название *конструктор по умолчанию*.

Конструктор копирования содержит один параметр – ссылку на объект этого же класса

```
Kvadrat(const Kvadrat &k) { /*тело конструктора*/ };
```

Если при описании класса не указать конструктор явно, как в описанном выше примере, то будет создан конструктор по умолчанию, значения полей при этом будут проинициализированы значениями по умолчанию соответствующих типов.

Приведем пример более детального описания класса `Kvadrat`. Класс включает в себя следующие элементы:

– одно вещественное поле, описывающее сторону квадрата;

– конструктор с одним вещественным параметром;

– конструктор по умолчанию, задающий сторону квадрата равной десяти;

– метод вычисления площади квадрата, возвращающий 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, так как 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; };
};//окончание описания класса Kvadrat
```

Для тех, кто читал внимательно, очевидно, что поле *a* объявлено с модификатором доступа `private`, а все методы, включая конструкторы, – с модификатором доступа `public`.

Так как 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, их нужно указать в круглых скобках после имени объекта, иначе будет вызван конструктор по умолчанию.

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

Создание объектов описанного выше класса `Kvadrat` должно выглядеть так:

`Kvadrat kv1;//` вызов конструктора по умолчанию;

`Kvadrat kv2(4);//` вызов конструктора с параметром.

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 создает 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, и если в этом случае не освободить эту память явно, то она будет считаться занятой даже после уничтожения объекта.

Пример описания деструктора.

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

Чтобы убедиться в работе деструктора без явного его вызова, удобно использовать отладочную печать. Например, так:

```
~Kvadrat() { cout << " Я - деструктор";};
```

Методы

При создании объекта в памяти будет отведено место под поля-данные, методы не дублируются, они создаются один раз при описании класса.

Если назначение метода не предполагает изменения значения полей объекта, то во избежание ошибок, связанных со случайным изменением значений полей, рекомендуется объявлять эти методы с ключевым словом `const`. Попытка изменения значений полей в теле такого метода приведёт к сообщению об ошибке.

Ниже приведено объявление метода вычисления площади квадрата, которое будет более правильным и безопасным с точки зрения работы с данными

```
float Sq() const { return a * a; };
```

Для того чтобы выполнился код метода, этот метод необходимо вызвать. Вызов метода для обработки полей какого-либо объекта происходит следующим образом: необходимо указать имя метода после имени объекта, для которого он вызывается, разделив их точкой. Так как метод – это функция, при его вызове после имени обязательно указать круглые скобки. Как и у любой функции, 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 так же, как возвращаемое значение любой функции.

Пример вызова метода, описанного выше класса:

```
Kv1.Set_a(kv2.Get_a());
```

В приведённой выше строчке происходит задание значения поля – стороны квадрата, описанного объектом *kv1*, равным значению стороны объекта *kv2*

```
Kvadrat2.Set_a(4.5);
```

Сторона квадрата, описанного объектом *kv2*, примет значение 4.5.

Выведем на экран значение площадей квадратов, описанных объектами *kv1* и *kv2*,

```
printf("S1=%f   S2=%f\n\n", kv1.Sq(), kv2.Sq());
```

В качестве параметра методу может быть передан объект того же или другого класса, например, если необходимо вычислить сумму одноимённых полей двух объектов. Опишем метод, возвращающий сумму площадей двух квадратов:

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

При разработке класса может возникнуть необходимость создания методов, вызов которых происходит без привязки к какому-либо

экземпляру класса. Такие методы называют статическими. При их объявлении необходимо указать ключевое слово `static` в заголовке метода. Статические методы могут обрабатывать только статические поля класса и вызывать другие только статические методы класса.

Статические поля класса предназначены для хранения информации, общей для всех объектов класса. Например, это может быть информация о количестве созданных объектов этого класса или указатель на область памяти, обращаться к которой могут все объекты класса.

Добавим в наш класс `Kvadrat` статическое поле `count` для хранения информации о количестве созданных объектов и два статических метода для работы с этим полем. Метод `Inc` будет увеличивать значение статического поля `count` при создании нового объекта, метод `Show_count` – выводить на экран значение поля `count`.

```
class Kvadrat
{static int count;
 float a;
 public:
    static void Inc() { count++; }
    Kvadrat(float A) { a = A; Inc(); };
    Kvadrat() { a = 10; Inc();};
    static void Show_count() { cout<<"count = "<<count;
}
    float Get_a() { return a; };
    void Set_a(float A) { a = A; };
    float Sq()const { return a * a; };

};// Kvadrat
```

Обратим внимание, что метод `Inc()` вызывается всякий раз при создании объекта, т. е. во всех описанных конструкторах. Статическая переменная-член класса должна быть создана и проинициализирована один раз вне функций. Создадим несколько объектов класса `Kvadrat` и выведем на экран информацию об их количестве с помощью метода `Show_count()`.

```
    int Kvadrat::count = 0;
    int main()
```

```
{Kvadrat k1,k2, *pKv= new Kvadrat;
    Kvadrat k3(7);
    Kvadrat::Show_count();
    delete pKv;
    Kvadrat::Show_count();
}//main
```

Указатель this

Для обращения к полям объекта в теле метода достаточно указать имя соответствующего поля. Объект, для которого вызван метод, передавать в метод не нужно. Он будет передан неявно через указатель `this.` Если в теле метода необходимо обратиться к полям ещё какого-либо объекта, его необходимо передать в метод в качестве параметра.

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 для обращения к полям объекта и даже к самому объекту, для этого указатель необходимо разыменовать. Например, если необходимо вернуть из метода сам объект, то следует писать `return *this`.

Перенастроить указатель `this` нельзя, это константный указатель.

Таким образом, обращение в теле метода к полям объекта, для которого вызван метод, происходит через переход по указателю `this`. В коде это обращение можно опустить, оно будет сгенерировано автоматически. Например, метод `Set_a`, приведённый в примере выше, можно описать следующим образом:

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

При этом компилятором будет сгенерировано обращение к полям соответствующего объекта через указатель `this`. `this-> a = A;`

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

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, статическое поле для подсчёта количества созданных объектов. Описать два конструктора: конструктор по умолчанию и конструктор с параметрами. Конструктор с параметрами должен содержать три аргумента: две стороны и угол между ними. Написать *Get*- и *Set*-методы для полей, методы вычисления третей стороны, периметра и площади треугольника. Предусмотреть своевременное вычисление третьей стороны при любом изменении одного из значений полей.

Создать несколько статических и динамических объектов описанного класса, статический массив объектов, набор объектов, используя массив указателей.

Вычислить сумму площадей всех созданных объектов.

Вывести на экран информацию о количестве созданных объектов, содержащуюся в статическом поле.

Указания к работе

1. В этой и всех последующих работах следует соблюдать определенные правила именования:

– все имена должны нести смысловую нагрузку;

– название классов и методов должны начинаться с прописной буквы;

– если имя класса или функции состоит из нескольких слов, каждое из них должно начинаться с прописной буквы;

– имена переменных, полей данных объектов должны начинаться со строчной буквы;

– если имя переменной состоит из нескольких слов, их следует разделять нижним подчёркиванием;

– имена полей данных начинаются со строчной буквы и заканчиваются нижним подчёркиванием;

– имена констант начинаются со строчной буквы *k*.

2. Класс треугольник описать в отдельном модуле, все необходимые объявления вынести в заголовочный файл.

3. Осуществить раздельную компиляцию проекта.

4. Предусмотреть своевременное вычисление третьей стороны при любом изменении одного из значений полей.

5. Для демонстрации работы деструктора наделить его отладочной печатью.

6. Функция `main()` должна реализовывать демонстрацию корректной работы всех методов класса.

Вопросы для закрепления материала

1. Что такое объект?
2. Что такое класс?
3. Что называют полями класса?
4. Какие поля называют статическими?
5. Что такое метод класса?
6. Какие методы называют статическими?
7. Для чего используют модификаторы доступа?
8. Какие вы знаете модификаторы доступа?
9. Что такое конструктор?
10. Какие виды конструкторов существуют?
11. Сколько конструкторов может быть описано в классе?
12. Что такое указатель *this*?
13. Может ли программист перенастроить указатель *this*?
14. Возможен ли доступ к указателю *this* из *main* функции?
15. Что такое деструктор?

Задания для самостоятельной работы

Все описываемые классы должны содержать конструктор по умолчанию, конструктор с параметрами, S*et*-, G*et*-методы для всех полей (если значение поля вычисляется на основе значений других полей, то описать только G*et*-метод). `main()` функция должна демонстрировать успешную работу всех методов класса.

1. Описать класс *Worker*, содержащий следующие *private*-поля: *name* (имя), *age* (возраст), *salary* (зарплата) и *public*-методы S*etName*, G*etName*, *SetAge*, G*etAge*, *SetSalary*, *GetSalary*. Создать два объекта этого класса со следующими значениями полей: имя – «Пётр», возраст – 17, зарплата – 20000 и «Иннокентий», возраст – 27, зарплата – 40 000. Вывести на экран сумму и разность зарплат Петра и Иннокентия, разницу в возрасте Петра и Иннокентия.

2. Дополнить класс *Worker* из предыдущей задачи *private*-методом *CheckAge*, проверяющим возраст на корректность (от 1 до 100 лет). Использовать метод *SetAge* для задания нового значения возраста (при попытке установить некорректное значение значение поля *age* должно остаться старым).

3. Описать класс *User*, содержащий следующие *protected*-поля: *name* (имя), *age* (возраст), *public*-методы *SetName, GetName, SetAge, GetAge.* Создать массив объектов класса *User*. Вычислить и вывести на экран средний возраст пользователей (элементов массива), вывести на экран имя самого молодого и самого старшего из пользователей, описанных объектами массива.

4. Описать класс *Student*, содержащий следующие *private*-поля: *name* (имя), *age* (возраст), курс, средний балл за всё время обучения, а также геттеры и сеттеры для всех полей объекта.

5. Создать набор из десяти объектов, вывести на экран информацию о студенте с самым высоким средним баллом и о студенте с самым низким средним баллом.

6. Описать класс *Driver*, содержащий следующие *private*-поля: *name* (имя), *age* (возраст), водительский стаж, категория вождения (*A, B, C*). Создать массив объектов описанного класса. Вывести на экран информацию о всех водителях с категорией, указанной пользователем.

7. Описать класс *Circle*, содержащий *private*-поле, *r* – радиус круга, методы вычисления площади и длины дуги окружности, статическое поле для хранения информации о количестве созданных объектов. Создать несколько статических и динамических объектов, вывести на экран информацию об их количестве и сумму площадей всех кругов. Удалить динамические объекты и снова вывести значение статического поля.

8. Добавить описание предыдущего класса полями для хранения координат центра круга методом масштабирования и методом проверки: принадлежит ли указанная точка данному кругу.

9. Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса, создания и уничтожения объектов этого класса. Используя статическое поле класса, вывести на экран информацию о количестве созданных объектов.

10. Описать класс *Rectangle*. Класс должен содержать два вещественных поля – длины сторон, одно статическое вещественное поле – сумму площадей всех созданных прямоугольников, методы масштабирования, вычисления периметра и площади. Создать набор из 10 объектов описанного класса. Объявить два указателя на класс *Rectangle*. Первый указатель настроить на элемент массива объектов описанного класса с наименьшей площадью, второй указатель настроить на элемент – объект с наибольшим периметром. Увеличить вдвое длины сторон объекта с самой маленькой площадью, уменьшить втрое значения сторон объекта с наибольшим периметром. К объектам обращаться через настроенные на них указатели. Вывести на экран сумму площадей указуемых объектов до и после масштабирования.

11. Описать класс прямоугольников со сторонами, параллельными осям координат. Прямоугольник задать длинами сторон и координатами левого верхнего угла. Написать методы перемещения прямоугольника на плоскости, масштабирования, построения наименьшего прямоугольника, содержащего два заданных прямоугольника, и построения прямоугольника, являющегося общей частью (пересечением) двух данных прямоугольников.

Дружественные функции

Внешнюю функцию, имеющую возможность доступа к скрытым членам класса, называют *дружественной классу функцией*.

Для того чтобы функция обладала этой возможностью, её необходимо объявить в классе с ключевым словом `friend`.

Так как дружественная функция – это внешняя функция, то она не получает в качестве скрытого параметра константный указатель `this`, поэтому объект, с которым планируется работать в функции, необходимо передать в такую функцию явно.

Пример заголовка дружественной функции:

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

Перегрузка операций

Для решения ряда практических задач часто требуется реализация операций сложения, вычитания, умножения и других на объектах класса. При этом удобно использовать привычные знаки операций для выполнения действий с одноимённым смыслом. Например, для сложения двух объектов использовать знак +, для деления – знак /.

При этом то, каким образом будет осуществляться само действие на объектах класса, определяет программист. В языке *C++* эта задача решается через механизм перегрузки функций-операторов. При этом сохраняется возможность использовать операторы в их привычном виде.

Важно помнить, что перегрузка нескольких операций: ., *., ?:, #, ##, `sizeof()` запрещена.

Для перегрузки оператора нужно использовать в заголовке функции в качестве её имени ключевое слово `operator` и знак соответствующей операции. При этом обязательно сохранение количества аргументов перегружаемого оператора. Приоритет операции при перегрузке также сохраняется.

Перегруженные операторы наследуются, за исключением операции присваивания. Перегрузить оператор можно как метод класса, как дружественную функцию и как внешнюю функцию. Перегруженные операторы – члены класса (методы) не могут быть статическими.

Перегрузка бинарных операторных функций

Если перегружается бинарная операция, необходимо учесть, что эта функция должна иметь два аргумента. Если речь идёт о методе, очевидно, что один параметр – левый операнд, т. е. объект, для которого планируется вызывать метод, будет передан в функцию неявно через указатель `this`. Второй аргумент – правый операнд – необходимо передать в метод явно.

Если бинарная функция перегружается как внешняя функция, то оба операнда необходимо передавать явно, так как дружественная 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); }
A operator + (A ob) {return A(a+ob.a);}
};
```

`operator+` имеет один явный формальный параметр – объект класса `A`, выступающий в роли второго слагаемого.

Первое слагаемое передаётся неявно через указатель `this`.

Перегруженный оператор возвращает 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();
```

Результат работы этого фрагмента кода приведен на рис. 1.

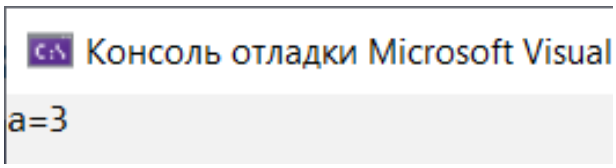


Рис. 1. Результат работы фрагмента программы

Операторные функции перегружают для использования их при вызове в привычном операторном виде, т. е. указывая только знак операции между операндами. Если перегруженный оператор является функцией-членом класса, возможен его вызов по правилам для обычных методов.

<имя объекта>.<имя метода>(<список фактических параметров>);

Для перегруженного выше оператора вызов будет выглядеть так:

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

Здесь:

`ob_A1` – первое слагаемое – объект, для которого вызван метод;

`ob_A2` – второе слагаемое – аргумент функции-члена;

`ob_A4` – вновь созданный объект, принимающий значение суммы объектов `ob_A1` и `ob_A2`.

Вывод на экран значения объекта `ob_A4` с помощью метода `Print()` приводит к результату, представленному на рис. 2, совпадающему с результатом, полученным в предыдущем примере. Таким образом продемонстрированы две возможности вызова перегруженного оператора, первый вариант вызова предпочтительнее.

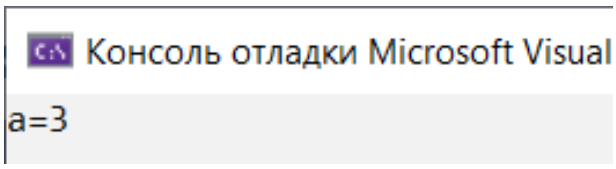


Рис. 2. Результат работы фрагмента программы

Опишем класс *B* и перегрузим в нём оператор + как дружественную функцию. Так как дружественная функция не имеет указателя `this`, ей необходимо передать два параметра (первое и второе слагаемые) 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-оператор, сохраним, а затем выведем на экран результат её работы

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

Результат работы приведённого выше фрагмента кода представлен на рис. 3.

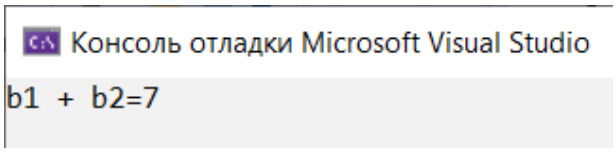


Рис. 3. Результат работы программы

Вернёмся к рассмотренному ранее классу Kvadrat. Дополним этот класс перегруженным методом operator+ и дружественной функцией operator-. Наделим эти функции следующей смысловой нагрузкой: сложение и вычитание соответственно площадей квадратов

```
float operator+ (Kvadrat ob) {return Sq() + ob.Sq();}
float friend operator- (Kvadrat ob1, Kvadrat ob2);
```

Дружественная функция должна быть описана за пределами класса следующим образом:

```
float operator- (Kvadrat ob1, Kvadrat ob2)
 {return ob1.Sq() - ob2.Sq();
```

Создадим два объекта класса Kvadrat и вызовем для них перегруженные операторы

```
Kvadrat k1, k2;
cout << "k1+k2=" << k1 + k2 << '\n';
cout << "k1-k2=" << k1 - k2 << '\n';
```

Создадим ещё один объект:

```
Kvadrat k3(7);
```

Попытка вычислить сумму трёх существующих объектов в одном выражении `k1 + k2 + k3` приведёт к ошибке. Так как сначала будет вычислена сумма объектов k1 и k2, а результат этого выражения – вещественное число, а не объект класса, следовательно, результат сложения двух объектов не может быть первым слагаемым – объектом, для которого вызывается перегруженный метод.

Для решения этой задачи перегрузим в классе ещё раз оператор + следующим образом:

```
float operator+ (float a) { return Sq() + a; }
```

Так как это метод, то первым слагаемым будет объект класса Kvadrat, для которого вызван метод, а вторым – вещественное число.

Используем в качестве параметра перегруженного оператора сумму двух объектов, для изменения порядка действий используем круглые скобки.

```
cout << "k1+k2+k3=" << k1 + (k2 + k3) << '\n';
```

Такое выражение будет вычислено корректно.

Вспомним, что при передаче объекта в функцию по значению создаётся его локальная копия при помощи конструктора копирования, сгенерированного автоматически. Значение статического поля count в этом конструкторе не увеличивается. При выходе за пределы функции

локальный объект будет уничтожен, при этом будет вызван деструктор, в теле которого уменьшается значение поля count. Таким образом, значение этого поля будет хранить некорректную информацию. Для избежания этой ошибки опишем явно конструктор копирования и увеличим в нём значение поля count.

Далее приведено полное описание класса Kvadrat и вызов вновь добавленных методов.

```
class Kvadrat
{
    static int count;
    float a;
public:
    static void Inc() { count++; }
    Kvadrat(float A) { a = A; Inc(); };
    Kvadrat() { a = 10; Inc();};
    Kvadrat(const Kvadrat &k) {  Inc(); };
    ~Kvadrat() { cout << " Деструктор" << " "<< --count
<< " об.\n";; };
    static   void   Show_count()   {   cout<<"count   =
"<<count<<"\n"; }
    float Get_a() { return a; };
    void Set_a(float A) { a = A; };
    float Sq()const { return a * a; };
    float  operator+  (Kvadrat  ob)  {  return  Sq()  +
ob.Sq(); }
    float operator+ (float a) { return Sq() + a; }
    float friend operator- (Kvadrat ob1, Kvadrat ob2);
};// Kvadrat
float  operator- (Kvadrat ob1, Kvadrat ob2) { return
(ob1.Sq() - ob2.Sq()); }
int Kvadrat::count = 0;
int main()
{
    Kvadrat k1, k2;
    Kvadrat k3(7);
    cout << "k1+k2=" << k1 + k2 << '\n';
    cout << "k1+k2+k3=" << k1 + (k2 + k3) << '\n';
    cout << "k1-k2=" << k1 - k2 << '\n';
```

Результат работы рассмотренного примера приведён ниже.

```
Деструктор 3 об.
k1+k2=100
 Деструктор 3 об.
k1+k2+k3=200
 Деструктор 4 об.
 Деструктор 3 об.
k1-k2=0
 Деструктор 2 об.
 Деструктор 1 об.
 Деструктор 0 об.
```

Отладочная печать, которая используется в теле деструктора, позволяет проследить за удалением объектов при выходе из функции.

Перегрузка унарных операторных функций

Если унарная функция перегружается как метод, то этот метод должен быть не статическим и не должен явно принимать параметры, так как единственный операнд унарной функции будет передан в метод неявно через указатель `this`.

Если унарный метод перегружается как внешняя функция, то эта функция должна иметь один передаваемый явно параметр – объект, к которому будет применён оператор.

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

Префиксная функция возвращает в точку вызова новое измененное значение, поэтому необходимо вернуть из функции объект, для которого она была вызвана, 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, так как 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 будет 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 представлен на рис. 4.

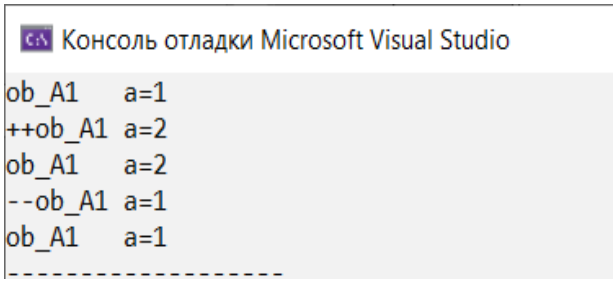


Рис. 4. Результат вызова перегруженных функций

Постфиксный инкремент (декремент) имеет такой же знак операции, как и префиксный. В этом случае нельзя по знаку операции определить, – какая это функция: постфиксная или префиксная. Для того чтобы постфиксные функции отличить от префиксных при объявлении и определении, в заголовке постфиксных функций указывают фиктивный целочисленный параметр. При вызове оператора этот 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 неизменённое значение операнда, а в дальнейшей работе будет использовано новое, изменённое значение.

Чтобы корректно перегрузить постфиксную операцию, необходимо в теле функции скопировать объект, для которого вызван метод, в локальную переменную и вернуть эту копию из функции. Значение текущего объекта, т. е. объекта, для которого вызван метод, должно быть изменено после того, как объект скопирован и до использования ключевого слова `return`. Так как объект передастся в метод через указатель, его значение изменится и будет доступно при дальнейшей работе.

Если постфиксная функция перегружена как внешняя функция, то объект, с которым планируется работать, необходимо передать в функцию явно. Причем передать объект нужно по ссылке, чтобы работа велась не с локальной копией, а с внешним объектом.

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 рис. 5.

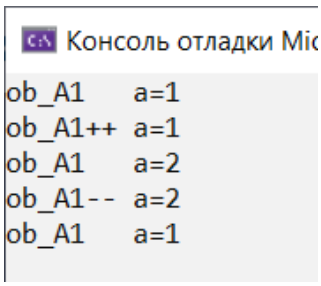


Рис. 5. Результат работы перегруженных
постфиксных функций для класса `A`

Из результатов работы видно, что в точку вызова вернулось старое значение, а в следующей строке в операторе вывода на экран было использовано новое значение.

Итак, при перегрузке постфиксной и префиксной функций необходимо учесть их отличие в возвращаемом значении и отличие при объявлении – фиктивный параметр в случае постфиксного оператора.

Перегрузите самостоятельно постфиксный и префиксный инкременты в классе Квадрат и напишите для этого класса дружественные функции – постфиксный и префиксный декременты.

Перегрузка операции присваивания

Операция присваивания, если она не описана в классе явно, будет сгенерирована автоматически. Операция присваивания используется для копирования полей одного объекта – источника (правый операнд) в поля объекта-приёмника (объект, вызвавший метод, – левый операнд).

При перегрузке операции присваивания в качестве возвращаемого значения нужно указать ссылку на объект, для которого вызван оператор.

Заметим, что работа операции присваивания, сгенерированной по умолчанию, была продемонстрирована в примере использования перегруженного оператора +. `A ob_A3=ob_A1+ ob_A2;`

Перегрузим операцию присваивания в описанном ранее классе A:

```
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();
```

На рис. 6 показан результат работы приведенного фрагмента кода, демонстрирующего работу перегруженного оператора присваивания.

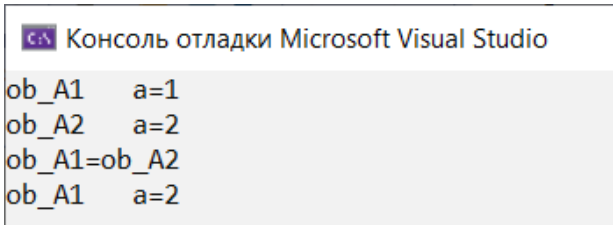


Рис. 6. Результат работы перегруженного оператора = в классе A

Как видно из рисунка, 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 можно видеть на рис. 7.

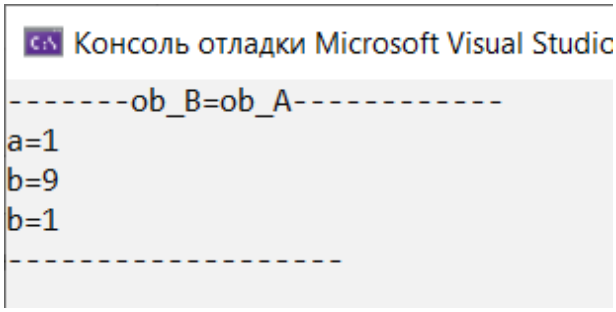


Рис. 7. Демонстрация работы перегруженного
в классе *B* оператора =

Лабораторная работа № 2
ДРУЖЕСТВЕННЫЕ ФУНКЦИИ. ПЕРЕГРУЗКА ОПЕРАТОРОВ

Цель работы

Изучение основ объектно ориентированного программирования, получение навыков создания дружественных функций перегрузки операций в классе.

Постановка задачи

Дополнить класс Треугольник, описанный в предыдущей работе, следующими методами: перегруженными операциями сложения, вычитания и умножения, постфиксной и префиксной формами инкремента и декремента. Одну бинарную и одну унарную операцию реализовать как дружественную функцию.

Описать в классе конструктор копирования, предусмотреть в нём изменение статического поля – количества объектов.

Продемонстрировать работу всех описанных методов на созданных объектах. Один из перегруженных операторов реализовать в виде дружественной функции.

Вывести на экран информацию о количестве объектов.

Указания к работе

Все перегруженные операции должны реализовывать действия, приводящие к результату, имеющему геометрический смысл. Например, возможен следующий подход. Операции сложения и вычитания возвращают сумму и разность площадей треугольников соответственно, операция умножения реализует масштабирование треугольника, операции декремента и инкремента изменяют значение угла между двумя сторонами; здесь важно не забыть пересчитать значение третьей стороны треугольника.

Вопросы для закрепления материала

1. Какие функции называют дружественными?
2. Что называют перегрузкой операции?
3. Можно ли перегружать операции для базовых типов данных?
4. Можно ли изменять количество параметров при перегрузке операции?

5. Можно ли перегрузить в классе одну операцию несколько раз?

6. Сколько и почему именно столько параметров нужно передать в метод класса, если он является перегруженным бинарным оператором?

7. Сколько и почему именно столько параметров нужно передать в метод класса, если он является перегруженным унарным оператором?

8. Сколько и почему именно столько параметров нужно передать дружественной функции, если она является перегруженным бинарным оператором?

9. Сколько и почему именно столько параметров нужно передать дружественной функции класса, если она является перегруженным унарным оператором?

10. Каким образом происходит отличие постфиксной от префиксной формы операторов инкремента и декремента в заголовке функции?

11. Что общего и в чём отличия дружественной функции и метода класса?

Задания для самостоятельной работы

1. Дополните класс *Worker* из задания 2 лабораторной работы № 1 перегруженной операцией сложения, которая позволит увеличивать зарплату работника.

2. Дополните класс *User* из задания 3 лабораторной работы № 1 перегруженными операциями постфиксного инкремента и декремента, которые позволят изменять возраст пользователя.

3. Дополните класс *Student* из задания 4 лабораторной работы № 1 дружественными функциями – перегруженными операциями постфиксного инкремента и декремента, которые позволят изменять возраст студента.

4. Дополните класс *Driver* из задания 5 лабораторной работы № 1 дружественными функциями – перегруженными операциями префиксного инкремента и декремента, которые позволят изменять стаж водителя.

5. 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.

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

7. 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.

8. 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.

9. Составить описание класса для определения одномерных массивов целых чисел (векторов). Предусмотреть возможность обращения к отдельному элементу массива с контролем выхода за пределы индексов, 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.

10. 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, печати (вывода на экран) элементов массива и всего массива.

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

Наследование

Теоретические сведения и методические указания к работе

Одиночное наследование

К механизму наследования в объектно ориентированном программировании прибегают при необходимости создать иерархию 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 возможно объявление дополнительных своих полей и методов. Методы базового класса в классе-наследнике могут быть перегружены.

При объявлении класса-наследника после его имени ставят двоеточие, затем указывают ключ доступа и имя базового класса

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

Объявим класс `C` как наследник класса `A`.

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

Вновь описываемый дочерний класс может быть наследником нескольких родительских классов. В этом случае в заголовке класса-наследника нужно перечислить через запятую все базовые классы, указав перед каждым родительским соответствующий ключ доступа.

Перепишем объявление класса `C` как наследника от классов `A` и `B`.

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

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

Ключ доступа, указываемый перед именем родительского класса, влияет на то, с каким модификатором будут унаследованы члены родительского класса. По умолчанию используют ключ доступа `private`.

В левой графе таблицы указан ключ доступа, используемый в средней графе, – спецификатор члена класса, с которым он объявлен в родительском классе, в правой графе – модификатор доступа, с которым член класса будет объявлен в классе-наследнике.

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

Обратите внимание, что `private` – члены класса не наследуются и не будут присутствовать в классе-наследнике ни при каком ключе наследования. Если закрытые от внешнего использования члены класса необходимо сделать доступными для наследования, то в базовом классе их объявляют с модификатором `protected`.

Члены базового класса, объявленные со спецификатором `public`, в классе-наследнике получат спецификатор доступа, соответствующий указанному ключу.

Правила наследования особых методов класса

Конструкторы, деструкторы, перегруженная операция присваивания не наследуются. Поэтому в классе-наследнике должны быть описаны собственные конструкторы и деструкторы и при необходимости операция присваивания.

При этом нужно учесть, что *вызов конструкторов выполняется в порядке наследования, а деструкторов – в обратном порядке.*

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.

Если в производном классе явно не описать деструктор, он будет сгенерирован автоматически. При вызове такого деструктора будет осуществлён вызов деструкторов всех базовых классов.

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

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

При написании методов класса-наследника нужно стараться избегать дублирования кода, уже описанного в методах родительского класса. Для решения этой задачи прибегают к вызову методов базового класса в теле методов наследника. Такой подход позволяет не только сократить код, но и упростить его модификацию.

При необходимости расширить или изменить функционал методов базового класса в наследнике, эти методы можно переопределить. При необходимости вызвать соответствующий метод базового класса для объекта производного класса, следует использовать четыре точки для уточнения имени.

Так, например, если класс `C` – наследник класса `A`. В классе `A` определён метод `print()`, в классе `C` этот метод переопределён. Создадим объект класса `C` и вызовем для него метод родительского класса

```
C ob1(1,2,3); ob1.print();
ob1.A::print();
```

Для объекта `ob1` вызван метод `print()` класса `A`.

Множественное наследование

Если класс-наследник имеет несколько родительских классов, то говорят о множественном наследовании. При этом если базовые классы имеют одноимённые члены, возникает конфликт имён, который устраняется с помощью оператора уточнения области видимости `::`. Этот оператор позволяет указать, в какой области видимости находится идентификатор.

Ниже приведён пример описания класса `C` с использованием множественного наследования и вызова методов для объектов этого класса.

```
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();
}
```

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

На рис. 8 показан результат работы приведённого выше фрагмента кода.

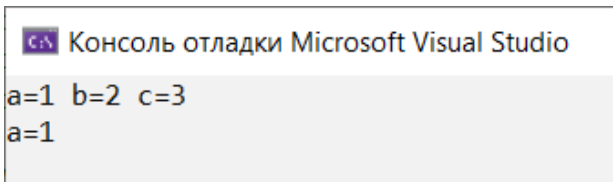


Рис. 8. Вызов метода базового класса для объекта класса-наследника

При создании иерархии классов возможна ситуация, при которой класс-наследник имеет несколько родительских классов с общим предком. В этом случае оба родительских класса могут иметь одинаковые поля. Таким образом, в классе-наследнике эти элементы будут унаследованы несколько раз. Чтобы этого избежать, общий предок объявляют виртуальным.

Лабораторная работа № 3
НАСЛЕДОВАНИЕ

Цель работы

Совершенствование навыков объектно ориентированного программирования, разработки классов, описания методов и дружественных функций. Изучение механизма наследования в языке программирования *C++*. Создание иерархии классов, переопределение методов.

Постановка задачи

Разработать класс Пирамида – наследник класса Треугольник, описанный в предыдущей лабораторной работе. Класс Пирамида, кроме унаследованных полей и методов, должен содержать одно дополнительное поле – высоту пирамиды, метод, возвращающий объём пирамиды. Поля, унаследованные от класса Треугольник, рассматриваются как основание пирамиды.

Описать дружественную классу Пирамида функцию, перегрузив для этого соответствующий оператор. Функция должна вычислять и возвращать в точку вызова разность или сумму (по желанию разработчика) объёмов двух пирамид. Продемонстрировать работу перегруженного оператора на объектах класса Пирамида.

Перегрузить оператор присваивания как метод класса Пирамида. Оператор должен реализовывать присвоение объекта класса Треугольник объекту класса Пирамида. В результате применения метода в основании пирамиды-приёмника должен оказаться Треугольник – объект-источник. Продемонстрировать применение этого метода. Для этого вывести на экран значения полей объекта-источника и объекта-приёмника до и после копирования.

Создать массив объектов класса Пирамида.

Продемонстрировать работу всех методов класса Пирамида на элементах созданного массива.

Вопросы для закрепления материала

1. В каких задачах в программировании используют наследование?
2. В каком случае наследование называют одиночным?
3. В каком случае наследование называют множественным?
4. Возможно ли наследование конструкторов?
5. В каком порядке происходит вызов конструкторов при создании объекта-наследника?
6. В каком порядке происходит вызов деструкторов при удалении объекта-наследника?
7. Для чего используют ключ наследования?
8. С каким модификатором доступа будет унаследован член класса-потомка, если в родительском классе он был объявлен с модификатором *private*, а наследование произведено с ключом *public*?
9. С каким модификатором доступа будет унаследован член класса-потомка, если в родительском классе он был объявлен с модификатором *protected*, а наследование произведено с ключом *public*?
10. С каким модификатором доступа будет унаследован член класса-потомка, если в родительском классе он был объявлен с модификатором *public*, а наследование произведено с ключом *private?*

Задания для самостоятельной работы

Во всех из приведённых ниже задач каждый описанный класс должен содержать конструктор по умолчанию и конструктор с параметрами, деструктор с отладочной печатью, выводящей на экран строку – название класса. Во всех задачах в `main` функции создать объекты всех классов иерархии, продемонстрировать работу всех описанных методов.

1. Описать иерархию из трёх классов: Точка, Линия, Прямоугольник. Все три класса должны содержать метод для перемещения объекта на плоскости, классы Линия и Прямоугольник должны содержать метод, предназначенный для сравнения площадей объектов.

2. Описать иерархию классов: Точка, Квадрат, Прямоугольник. Все три класса должны содержать метод для перемещения объекта на плоскости. В классах Квадрат и Прямоугольник описать методы для определения пересечения объектов и определения включения одного объекта в другой.

3. Описать базовый класс Автомобиль и класс-наследник Автобус. В базовом классе описать поля – координаты и угол, задающий направление движения, метод: перемещения в заданном направлении на определённое расстояние и поворота на заданный угол. Класс Автобус добавить полями, содержащими число пассажиров и количество полученных денег за рейс и изначально равными нулю. Описать методы изменения числа пассажиров при входе и выходе из автобуса.

4. Описать базовый класс Животное, содержащий два приватных поля – строку для хранения клички и числовое поле для хранения веса; описать метод вывода на консоль сообщения с указанием имени и веса объекта класса Животное. Создать два производных класса: Домашнее животное и Дикое животное, в каждом из которых описать дополнительные поля и методы.

5. Описать базовый класс Калькулятор, имеющий одно скрытое поле – вещественное число x (левый операнд), и следующие методы: сложения с вещественным числом, вычитание, умножение, деление, извлечение квадратного корня, тригонометрические функции. Описать класс-наследник, имеющий дополнительное поле для хранения полученного результата, и методы записи и извлечения значения из памяти.

6. Описать класс Треугольник, имеющий три поля – стороны треугольника, методы проверки на существование треугольника, вычисления периметра и площади вывода на экран сведений о треугольнике.

Описать класс-наследник Равносторонний треугольник, перегрузить метод поверки существования треугольника на проверку, является ли треугольник равносторонним.

7. Описать родительский класс Служащий и производный класс Директор, обладающий более широким функционалом.

8. Описать родительский класс Продукт с тремя полями данными: название, цена, вес. Описать класс-наследник Покупка, содержащий данные о количестве покупаемого товара, его весе, и метод, выводящий эту информацию на экран.

9. Описать базовый класс Транспорт со следующими характеристиками: скорость, вес, запас топлива, пробег и другими, методами чтения и установки значений полей и другими возможными методами (не менее трёх). Описать два класса-наследника от класса Транспорт – Автомобиль и Самолёт, дополнив их всеми необходимыми полями и методами.

10. Описать базовый класс Человек с атрибутами: имя, год рождения, пол, вес. Создать класс-наследник Студент с дополнительными полями: специальность, курс, массив оценок, наличие задолженностей и т. п. Дополнить этот класс методами – средний балл, проверка наличия задолженностей.

Полиморфизм

Теоретические сведения и методические указания к работе

Работу с объектами классов часто организуют с использованием указателей.

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

Вызов методов будет производиться в соответствии с типом указателя, а не с типом объекта, на который он настроен, что не всегда удобно. Это обусловлено тем, что ссылки на методы проставляются во время компоновки программы. Этот процесс называется *ранним связыванием.*

В качестве подтверждения вышесказанного объявим объекты классов `A, B, C,` описанных ранее

```
A ob_A(7);
B ob_B(6);
C ob_C;
```

Теперь объявим указатели на объекты соответствующих классов и настроим их на созданные объекты в соответствии с их типами

```
A* pA = &ob_A;
B* pB = &ob_B;
C* pC=& ob_C;
```

Вызовем через проинициализированные указатели метод `print();`

Вызов метода через указатель производится следующим образом:

<указатель> -> <имя метода>([<список фактических параметров>]),

где операция -> производит два действия: разыменовывание и уточнение имени.

Вызовем метод `print()` двумя способами: через операцию `->` и непосредственно через операцию разыменовывания * и уточнение имени

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

Из результатов, представленных на рис. 9, видно, что, как и ожидалось, для каждого объекта был вызван метод соответствующего класса.

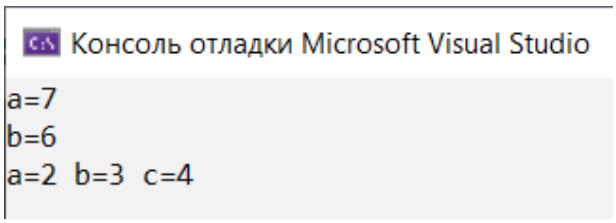


Рис. 9. Результат вызова метода через указатель на объект

Перенастроим указатель на родительский класс `pA` на объект класса-наследника и снова вызовем метод `print()`

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

Из результатов работы, представленных на рис. 10, следует, что для объекта класса `C` был вызван метод `print()` родительского класса, т. е. вызов произошёл по типу указателя.

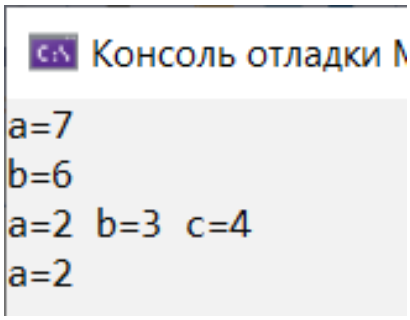


Рис. 10. Результат вызова метода родительского класса для объекта класса-наследника

Указатель на класс-наследник нельзя настроить на объекты базового класса. Если допустить такую возможность, то можно было бы вызывать методы наследника для объектов родительского класса, но в родительском классе может не быть полей, обрабатываемых методами дочернего класса. Следовательно, при таком подходе возникали бы ошибки во время исполнения программы, поэтому такая настройка указателей запрещена.

При необходимости осуществить вызов метода через указатель по типу указуемого, а не по типу настроенного на него указателя, используют механизм *позднего связывания*. При задействовании этого механизма, используя один указатель, можно вызывать методы разных классов иерархии в зависимости от того, на объект какого класса настроен этот указатель. При реализации позднего связывания ссылка на метод происходит в момент его вызова при исполнении программы. Обращение происходит к методу класса, соответствующему объекту, на который настроен указатель.

Процесс позднего связывания реализован при помощи виртуальных методов. Такой метод содержит в своём заголовке ключевое слово `virtual`.

```
virtual void metod(int a);
```

Виртуальные методы наследуются. Если метод определён в родительском классе как виртуальный, то и в классе-наследнике такой метод будет виртуальным. При переопределении виртуального метода в классе-потомке без ключевого слова `virtual` переопределённый метод не будет виртуальным.

Если виртуальный метод переопределён в классе-наследнике, то соответствующий метод базового класса для объекта класса-наследника можно вызвать, используя операцию доступа к области видимости `::`.

Если в классе содержится хотя бы один виртуальный метод, то для этого класса во время компиляции будет создана таблица виртуальных методов. В этой таблице в порядке объявления хранятся адреса виртуальных методов этого класса.

Каждый объект такого класса имеет дополнительное поле – ссылку на таблицу виртуальных методов (*vptr*). Во время компиляции ссылка на виртуальные методы заменяется на обращение к таблице виртуальных методов (*vtbl*). Обращение происходит через указатель *vptr*, хранящий адрес таблицы соответствующего класса. Во время исполнения программы адрес метода выбирается из таблицы.

Таким образом, из-за дополнительного этапа получения адреса вызов виртуального метода происходит медленнее, чем обычно, именно поэтому нет смысла объявлять виртуальными все методы класса.

Рекомендуется при работе через указатели объявлять виртуальными деструкторы, такой подход обеспечивает корректное освобождение памяти.

Очевидно, что механизм позднего связывания можно использовать только при работе с объектами через указатели. В таком случае говорят о *полиморфизме*. Проявление полиморфизма заключается в вызове различных методов, а следовательно, в осуществлении различных действий при выполнении одинаковых строчек кода.

Продемонстрируем вышеизложенные теоретические сведения на простом примере.

Внесём изменения в объявление метода `print` класса `A`, описанного в примерах ранее, сделав этот метод виртуальным.

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

Теперь при вызове метода `print` для объекта класса-наследника через указатель на родительский класс будет вызван метод класса указуемого.

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

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

Результат работы этого фрагмента программы, представленный на рис. 11, отличается от результатов, приведенных на рис. 10. Однако можно заметить, что изменений в настройке указателей и вызове методов нет.

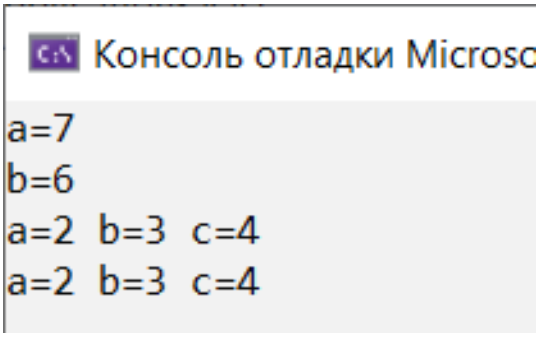


Рис. 11. Результат вызова полиморфного метода
через указатель на родительский класс

Понятно, что разница в результатах объясняется тем, что в последнем примере метод `print` виртуальный.

Виртуальный метод должен быть определён. Если нет необходимости в его реализации в базовом классе, реализация нужна только в потомках, такой метод базового класса должен содержать признак нуля вместо тела. Данный метод называют *чисто виртуальным*.

Класс, содержащий хотя бы один чисто виртуальный метод, называется *абстрактным.* Такие классы используют для описания общих представлений о классе. Абстрактные классы выступают в роли базовых классов иерархии, реализация методов происходит в классах-наследниках. Объект абстрактного класса создать нельзя, попытка вызова чисто виртуального метода приведет к ошибке.

Ниже приведён пример абстрактного класса с чисто виртуальным методом.

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

Лабораторная работа № 4.
ПОЛИМОРФИЗМ

Цель работы

Совершенствование навыков объектно ориентированного программирования, реализации наследования. Изучение механизмов раннего и позднего связывания, полиморфизма.

Постановка задачи

Дополнить класс Треугольник, описанный в предыдущей лабораторной работе, виртуальным методом `Show()`. Метод должен реализовывать следующий функционал: отображение на экране полной информации об объекте: значение сторон треугольника, угла между двумя заданными сторонами, периметра и площади. В производном классе Пирамида переопределить метод `Show()`, расширив его функционал, добавив отображение на экране высоты и объёма пирамиды.

Создать указатель на объект класса Треугольник и указатель на объект класса Пирамида, настроив их на объекты соответствующих классов.

Используя объявленные указатели, вызвать метод `Show()` для объектов соответствующих классов.

Осуществить перенастройку указателей на объекты несоответствующих классов: базовый – на производный, производный на родительский. Одно из этих действий приведёт к ошибке. Закомментировать строчку с ошибкой и пояснить в комментариях причину возникновения ошибки.

Вызвать метод `Show()` и объяснить полученный результат.

Создать набор из объектов классов Треугольник и Пирамида, вызвать метод `Show()` для всех объектов набора.

Вопросы для закрепления материала

1. Какие функции называют виртуальными?
2. В каком случае функцию объявляют виртуальной?
3. На каком шаге происходит раннее связывание?
4. На каком шаге происходит позднее связывание?
5. Что хранит таблица виртуальных методов?

6. Каким образом осуществляется связь объекта с нужной таблицей виртуальных методов?

7. В каком порядке расположены адреса в таблице виртуальных методов?

8. В каких случаях нецелесообразно задействование механизма позднего связывания?

9. В чём недостатки позднего связывания?

10. Метод какого класса будет вызван, если вызов осуществляется для объекта производного класса через указатель на базовый класс, а вызываемый метод – невиртуальный, унаследован и переопределён в производном классе?

Задания для самостоятельной работы

Во всех задачах для самостоятельной работы в `main()` продемонстрировать работу всех методов описанных классов. Все поля и методы должны иметь информативные имена и нести смысловую нагрузку в соответствии с назначением класса.

1. Описать иерархию классов: Учащийся, Ученик, Студент, Работающий студент. Наделить классы соответствующими данными и методами по усмотрению разработчика так, чтобы каждый следующий класс иерархии содержал данные родительского класса и свои дополнительные. Среди методов в каждом классе должен быть виртуальный метод – метод вывода информации об объекте на экран.

2. Описать базовый класс Фигура и производный класс Эллипсоид со следующими характеристиками: координаты *x, y, z*, унаследованные от класса Фигура, и собственные поля *a, b, c* – полуоси. В классе Эллипсоид переопределить функцию для расчета объема. Объем эллипсоида вычисляется по формуле $V = 4\pi abc/3$.

3. Описать абстрактный базовый класс Фигура, содержащий чисто виртуальный метод Площадь. Создать классы, производные от класса Фигура: Прямоугольник, Круг, Прямоугольный треугольник, Трапеция. Во всех производных классах переопределить метод вычисления площади в соответствии с назначением класса. Объявить массив указателей на абстрактный класс, настроить элементы массива на объекты производных классов. Вызвать метод вычисления площади, используя все элементы массива.

4. Разработать иерархию классов. Иерархия должна содержать не менее трёх уровней и не менее четырех классов (включая базовый). 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 описать конструктор. Все классы должны иметь смысловую нагрузку, т. е. описывать реально существующие объекты. Продемонстрировать работу с объектами классов через указатели, в том числе на базовый класс.

5. Описать абстрактный базовый класс для фигур на декартовой плоскости. Создать на основе этого класса иерархию классов, описывающих объекты на плоскости: Точка, Круг, Квадрат, Прямоугольник, Равнобедренный треугольник. Все классы должны содержать как минимум два виртуальных метода: метод, вычисляющий и возвращающий площадь объекта, и метод, отображающий на экране всю информацию о соответствующем экземпляре. Создать массив указателей на базовый класс, продемонстрировать работу с объектами всех классов через созданные указатели.

6. Создать базовый класс Массив с виртуальными методами сложения и поэлементной обработкой массива. Описать на основе этого класса производные классы Пересечение и Объединение. В классе Пересечение операция сложения реализуется как пересечение множеств, а поэлементная обработка представляет собой извлечение квадратного корня. В классе Объединение операция сложения реализуется как объединение, а поэлементная обработка – вычисление факториала элемента.

7. Описать иерархию классов для определения шахматных фигур. Абстрактный класс Шахматная фигура, содержащий поля – название, цвет и координаты позиции на доске. Производные классы должны содержать конструктор копирования, оператор присваивания для разнотипных фигур, метод для изменения координат на доске в соответствии с типом фигуры. Продемонстрировать работу с объектами классов через массив указателей на базовый класс.

8. Создать класс *3D*-фигура и производные классы Шар, Конус, Цилиндр и Куб. Каждый класс должен содержать следующие виртуальные методы: вычисление объёма и вывод на экран информации об объекте. Объявить массив указателей на базовый класс, настроить элементы массива на разнотипные объекты. Используя эти указатели, продемонстрировать работу всех методов иерархии.

9. Описать абстрактный класс Млекопитающие и два производных класса: Животные и Люди. От класса Животные определить два производных класса: Собака и Корова. В классах определить виртуальные функции, выводящие на экран информацию об объектах. В `main` функции организовать возможность создания объекта класса, который запросит пользователь.

10. Описать абстрактный базовый класс Норма с виртуальной функцией вычисления нормы и модуля. Создать производные классы Комплекс и Вектор, в которых переопределить функции 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 – корень квадратный из суммы квадратов координат; норма вектора – максимальное из абсолютных значений координат). Продемонстрировать работу со всеми методами описанных классов через указатели на соответствующие объекты.

Шаблоны классов

Теоретические сведения и методические указания к работе

Если необходимо разработать общую структуру класса, а в дальнейшем иметь возможность создавать экземпляры этого класса, обладающие одинаковыми по смыслу атрибутами и поведением, но с различными типами параметров, описывают шаблон класса.

Шаблоны используют при разработке контейнерных классов, т. е. классов, предназначенных для хранения данных определённым образом с возможностью доступа к отдельным элементам.

Описание класса-шаблона начинается с ключевого слова `template`.

Синтаксис описания шаблона следующий.

`template` *<описание параметров шаблона> определение класса*`;`

Параметры шаблона – это те типы данных, которые будут определены в момент создания объекта. Если параметров несколько, их перечисляют через запятую. Это могут быть базовые или пользовательские типы данных. Так как параметр класса 336 – это тип данных, то внутри класса он может быть употреблён в любом месте, где синтаксически необходимо указание типа. Область действия параметра шаблона – от точки его описания до конца описания шаблона класса.

Объявим шаблон класса контейнера

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

При создании объекта класса, описанного при помощи шаблона, необходимо после имени класса перечислить в угловых скобках аргументы класса в порядке, указанном при объявлении класса,

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

Для шаблона, описанного выше, создание объекта будет выглядеть так:

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

Объект `T_S1` предназначен для хранения пяти целых чисел.

Имя класса совместно с переданными аргументами можно рассматривать как уточнённое имя шаблона. Вызов методов для объектов класса, описанного как шаблон, осуществляется по тем же правилам, что и для обычных классов. Например, пусть `T_S_Kv` – объект класса контейнер, в который помещены объекты класса Квадрат, тогда строка `(T_S_Kv.POP()).Sq());` реализует вызов метода `Sq()` для объекта, извлечённого из вершины стека.

Работа с данными, извлеченными из хранилища, ведётся по тем же правилам, что и с любыми другими данными.

Создадим объект класса `Kvadrat` и скопируем в него элемент, извлеченный из хранилища,

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

Выведем на экран информацию о полученном объекте

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

Использование шаблонов классов позволяет эффективно задавать общие возможности для хранения и обработки данных разных типов. Однако нужно помнить, что программа, использующая шаблоны, содержит полный код всех порождённых классов.

Контейнерные классы

Очевидно, что контейнерные классы предназначены для хранения данных. Стандартная библиотека классов *C++* уже содержит определённый набор классов, позволяющих хранить данные, реализовывать доступ к их элементам и ряд других операций. Наиболее часто используются классы: векторы (`vector`*)*, очереди (`queue`), разновидности списков (`list`), очереди с приоритетом (`priority_ queue`).

При решении конкретных задач выбор класса контейнера обусловливается назначением хранимых данных и действиями, которые планируется с ними осуществлять.

Так, например, объекты класса `Vector` стоит использовать, если в работе предполагается произвольный доступ к элементам, вставка и удаление элемента в конец вектора.

Двустороннюю очередь используют, если кроме произвольного доступа к элементам предполагается вставка и удаление элементов с двух концов.

Списки (`List`) позволяют осуществлять вставку и удаление элементов из произвольной части хранилища, при этом произвольный доступ к элементам запрещён.

В приведённых ниже примерах продемонстрирована работа с объектами классов, о которых шла речь.

Для создания объектов соответствующих классов нужно подключить заголовочные файлы, содержащие объявления классов:

```
#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 примере создан объект класса `vector` для работы с целыми числами, в цикле `for` в хранилище добавлены значения, равные значению индекса счётчика цикла. На экран выведено значение элемента с номером 3, соответственно и его значение будет равно трём.

Теперь рассмотрим работу с объектом класса list.

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

Заполним список в цикле с помощью метода push_back()

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

Создадим итератор I и настроим его на начало списка, итератор можно рассматривать как указатель на элемент контейнера.

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

Теперь вставим в наш список элемент со значением -10, предварительно сдвинув итератор,

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

Снова настроим итератор на начало списка и выведем все значения списка на экран

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

Результат будет таким: 0 -10 1 2 3 4

Метод sort позволяет упорядочить элементы списка. Снова выведем на экран

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

На экране будут отображены элементы из контейнера, упорядоченные по возрастанию: -10 0 1 2 3 4

Создадим объект класса контейнера deque

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

Заполним его квадратами номеров элементов начиная с единицы.

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

В следующем цикле будем выводить элементы на экран и извлекать их из списка.

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

Если стандартного функционала недостаточно и возникает необходимость в описании пользовательского класса контейнера, необходимо предусмотреть два аспекта: создание самого объекта и выделение места под блок хранимых данных.

Если предполагается, что поля объекта и хранилище данных находятся в разных местах, то среди полей класса обязательно должно быть поле-указатель на вершину хранилища. Захват памяти под хранилище данных и настройка указателя должны быть осуществлены в конструкторе.

Пример объявления такого класса:

```
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; };
```

Проверка хранилища на пустоту или на заполненность должна осуществляется каждый раз при извлечении или добавлении элементов в хранилище.

Рекомендуется такую проверку выполнять методами, возвращающими логическое значение. Например, метод проверки контейнера на заполненность возвращает значение `true`, если он полностью заполнен. Для получения информации о заполненности хранилища достаточно проанализировать значение поля, хранящего номер текущего элемента.

Нумерацию элементов хранилища организуют таким образом, что в текущий момент времени значение поля-индекса равно номеру элемента, который будет помещён в хранилище следующим. Таким образом, если значение этого поля равно максимально возможному количеству элементов в хранилище, то контейнер полон, и наоборот, если значение поля-индекса равно нулю, стек пуст.

Примеры реализации таких функций:

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

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

Для реализации функционала класса контейнера необходимо разработать методы добавления и извлечения из него элементов. 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];}
```

Рассмотрим простой пример работы с объектом класса контейнера. Для этого создадим объект класса `Stack`, фрагменты описания которого были приведены выше, заполним хранилище данными и извлечём их

```
Stack St(5);
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 показан на рис. 12.

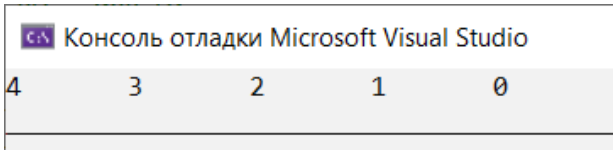


Рис. 12. Извлечение элементов из стека

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

При правильной организации работы с данными попытка добавить в заполненное хранилище элементы или извлечь из пустого хранилища не повлечёт за собой ошибку, а приведёт к появлению предупреждающего сообщения. Попробуем добавить в описанное хранилище больше возможного числа элементов. Перепишем предыдущий пример, изменив количество итераций в цикле `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 работы с изменением показан на рис. 13.

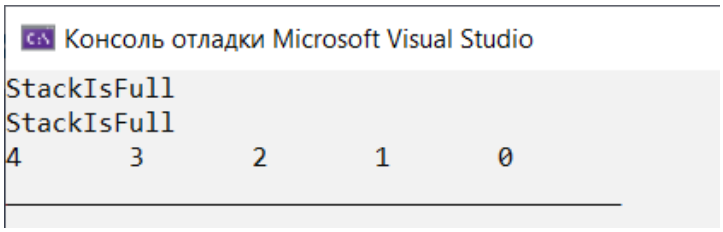


Рис. 13. Результат попытки переполнения стека

Из рисунка видно, что количество и порядок элементов в стеке не изменились. Попытка записи в заполненное хранилище привела к отображению на экране соответствующего сообщения.

Возможен другой подход к организации хранилища: совместное хранение объекта (т. е. его полей) и самого хранилища данных. Для реализации такого подхода в классе следует перегрузить оператор `new`. При создании динамического объекта сначала вызывается оператор `new`, выделяющий динамическую память, а затем – конструктор. Оператор `new,` перегруженный в классе, становится `static`-методом независимо от того, использован модификатор `static` или нет.

Заголовок метода будет выглядеть так:

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

Оператор `new` получает два параметра: размер объекта, количество элементов в хранилище и возвращает нетипизированный указатель. В теле метода необходимо захватить память в количестве, необходимом под поля объекта, и максимальный размер хранилища

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

Лабораторная работа № 5
КЛАССЫ КОНТЕЙНЕРЫ

Цель работы

Совершенствование навыков объектно ориентированного программирования. Получение опыта разработки контейнерных классов, описания и использования шаблонов классов.

Постановка задачи

Описать шаблон класса `Stack`, позволяющий организовать работу с данными по принципу стека.

Указания к работе

В разрабатываемом классе необходимо объявить поле-указатель на вершину стека. Хранилище данных организовать как динамический массив.

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

Для всех объектов заполнить хранилище, извлечь значения и вывести их на экран. Продемонстрировать корректную работу программы в случае попытки извлечь из пустого хранилища и попытки добавить в полностью заполненное хранилище. Для объекта, заполненного экземплярами класса Квадрат (или Круг), вычислить суммарную площадь всех объектов хранилища, используя метод соответствующего класса. Результат вычислений вывести на экран.

Вопросы для закрепления материала

1. Какие классы называют контейнерами?
2. Что такое шаблон класса?
3. В каких ситуациях применяют шаблоны классов?
4. Какими способами можно организовать взаимное расположение хранилища данных и управляющего объекта?
5. Что необходимо сделать для создания объекта контейнерного класса, описанного как шаблон, для заполнения его данными определённого типа?

6. Каким образом можно определить количество байт, которое необходимо выделить под объект, если управляющая часть объединена с хранилищем?

7. Что необходимо сделать для проверки контейнера на пустоту?

8. Чему будет равно значение поля-индекса, если класс-контейнер заполнен полностью?

9. Если контейнерный класс организован по принципу стека через массив, каким образом осуществляется доступ к элементам?

10. Как получить адрес первого элемента хранилища, если оно совмещено с управляющей частью объекта?

Задания для самостоятельной работы

Для всех перечисленных ниже вариантов разработать шаблон класса контейнера. Описать методы добавления и извлечения элементов, проверки на пустоту и на заполненность. Создать объекты-контейнеры для данных разных типов – базовых и пользовательских. Продемонстрировать работу всех методов класса.

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

Раздел 2
РАБОТА В ВИЗУАЛЬНОЙ СРЕДЕ ПРОЕКТИРОВАНИЯ

Объектно ориентированное программирование на языке C#

Теоретические сведения и методические указания к работе

Дальнейшее изучение курса основано на объектно ориентированном языке программирования C#. В прил. 4 – 6 приведена краткая информация по изучению языка программирования C#. Представленных сведений достаточно для освоения курса и дальнейшего более глубокого изучения языка C#. Основополагающие свойства объектно ориентированного подхода – инкапсуляция, наследование, полиморфизм – остаются неизменными.

Разберём их применительно к языку программирования C#.

Объединение в одной сущности данных и способов их обработки, исключая доступ извне, называют *инкапсуляцией*. Такой подход даёт возможность скрывать детали реализации от пользователя, обеспечивая защиту данных.

Для этого данные (поля или атрибуты) объекта должны быть объявлены с использованием ключевого слова `private` или `protected`, если предполагается наследование.

Описав класс, наделив его необходимыми атрибутами и поведением, можно создать нужное число экземпляров класса – объектов. Таким образом, класс можно рассматривать как пользовательский тип данных, объекты – переменные этого типа.

Рассмотрим в качестве примера описание с помощью класса всем известного предмета, часто используемого в повседневной жизни – шариковой ручки. Очевидно, что ручка обладает следующими свойствами: длина, диаметр шарика, цвет пасты, механическая или нет, наличие колпачка, цвет и материал корпуса, фирма-изготовитель. Для более точного описания можно дополнить список другими свойствами. Список методов также можно детально проработать, однако понятно, что стоит начать с основного свойства – свойства, для реализации которого ручка создана, – оставлять след на бумаге.

Описание класса на языке программирования C# начинается с указания имени класса после ключевого слова `class`. Далее следует описание класса, заключённое в фигурные скобки.

```
class PEN
{. . .
}
```

Предпочтительно описывать класс внутри какого-либо пространства имен. Правило создания объектов остаётся неизмененным, т. е. таким же, как для создания переменных любого типа.

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

Объекты в *C#* относятся к ссылочным типам, поэтому инициализация объекта будет выглядеть так:

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

Объявление и инициализацию переменной типа класс можно совместить

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

Модификатор доступа в C# может быть установлен не только для членов класса, но и для самих классов. Влияние модификаторов доступа на видимость членов класса остаётся таким же, как в C++.

`public` – член класса 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.

Свойства

Понятие свойства в языке C# несколько шире, чем в C++. Если в языке C++ свойства – это данные – члены класса, то в C# свойство можно рассматривать как совокупность поля, которое должно быть объявлено с модификатором `private` или `protected`, и его аксессоров, позволяющих читать и задавать значение поля. При этом в классе должна присутствовать строка объявления как поля, так и свойства. Если предполагается работа со свойством за пределами класса, то такое свойство должно быть открытым, т. е. его описание должно начинаться с ключевого слова `public`, затем указывают тип данных и имя свойства.

Поля принято именовать со строчной буквы, а свойства – с прописной.

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

После объявления свойства в фигурных скобках следует реализация аксессоров, позволяющих организовать доступ к значению свойства на чтение (`get`) и запись (`set`).

Аксессор `get` должен возвращать значение, для этого используют оператор `return`, указывая после него значение свойства. В рассматриваемом примере это будет значение поля `dlina`.

Аксессор `set` даёт возможность установить новое значение свойства. Для задания значения используют переменную `value`. Это виртуальная переменная, объявлять которую не нужно, она всегда существует внутри фигурных скобок после ключевого слова `set` и имеет такой же тип, как и свойство.

Если свойство должно быть только для чтения, аксессор `set` писать не нужно

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

При задании значения свойства стоит проводить его проверку на попадание в нужный диапазон

```
private int Dlina
{
get {return dlina;}
set { if (value>0 && value <50) dlina=value;}
}
```

По умолчанию аксессоры имеют модификатор доступа `public`.

Если объявить аксессор с модификатором `private`, то мы получаем свойство, обращаться к которому можно только внутри класса. При обращении к свойству из другого класса необходимо указывать полное имя, начав с имени объекта,

<имя объекта>.<имя свойства>

Внутри класса имя объекта писать не нужно.

Если в аксессорах не планируется осуществлять никаких других действий, кроме чтения и записи, можно использовать сокращённый вариант объявления свойства

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

Методы

Метод – это функция-член класса, предназначенная для работы с полями-данными класса. Как любая функция метод может возвращать в точку вызова значение, и если это не закрытый метод, то его можно вызвать за пределами класса, соответственно и возвращаемое значение будет доступно за пределами класса. Круглые скобки после имени метода обязательны, они могут быть пустыми, если метод не получает входных параметров.

Перед именем метода указывают модификатор доступа и тип возвращаемого значения. Если метод ничего возвращать не должен, пишут `void`. Далее в фигурных скобках следует реализация метода, т. е. код, который будет выполнен при его вызове. Фигурные скобки обязательны, даже если тело метода состоит из одного оператора.

Синтаксис объявления и описания метода:

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

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

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

Статические (или статичные) методы и данные класса объявляются с ключевым словом `static`. Эти поля и методы не связаны конкретным объектом, а относятся ко всему классу. Если в классе есть статическое

поле, оно создаётся один раз, сколько бы ни было создано объектов. Статический метод класса может быть вызван без создания объекта.

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 в классе не описан явно, в классе будет существовать конструктор по умолчанию. В приведённых выше примерах при создании объекта был вызван именно конструктор по умолчанию.

Конструктор, как любую функцию, можно перегрузить необходимое число раз. Использование различных конструкторов позволяет создавать объекты с различными начальными значениями атрибутов.

Перегруженный конструктор по умолчанию:

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

Опишем класс `Kvadrat`, содержащий одно поле – сторона квадрата 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. Такой подход используют при работе над проектом нескольких разработчиков.

Деструктор, или метод завершения

Класс может иметь только один метод завершения, поэтому эти методы нельзя наследовать и перегружать. Запускаются они автоматически, вызвать вручную их нельзя. Такой метод не имеет параметров.

Описание деструктора на языке C#, так же как и на языке *C++,* начинается с символа тильда (~), далее следует имя класса. Код, необходимый для освобождения памяти, генерируется автоматически.

Так как деструктор невозможно вызвать вручную, действия, которые необходимо 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. Статические методы не получают адреса объекта, поэтому все операнды должны быть переданы в метод в явном виде в качестве параметров и как минимум один из параметров должен быть типа соответствующего класса.

Перегрузим оператор+ в классе `Kvadrat`. Метод будет возвращать сумму площадей двух квадратов

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

При перегрузке операций инкремента и декремента не нужно перегружать отдельно префиксную и постфиксную формы. Перегрузку нужно осуществить один раз, возвращая значение нового объекта с изменёнными значениями полей, сам объект при этом изменять не нужно.

При операции постфиксного инкремента сначала будет создана временная переменная, в которой будет сохранён текущий объект. В качестве результата операции возвращается значение временной переменной. Затем значение объекта будет заменено на новое значение. При префиксном инкременте компилятор возвращает новое значение, полученное из функции:

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

Наследование

В языке *C#* допустимо только одиночное наследование. Если класс объявлен с модификатором `sealed`, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников.

Имя родительского класса указывают после имени класса-потомка через двоеточие

```
class < имя класса >: <имя родительского класса>
public class Rect : Kvadrat
{…
}
```

Класс Rect – наследник класса Kvadrat – будет содержать унаследованное свойство Storona, если в классе оно было описано с модификатором protected

```
protected float storona;
```

Для корректного описания прямоугольника в классе необходимо описать ещё одно свойство (другую сторону) и переопределить метод вычисления площади. Заголовок переопределяемого свойства начинается с ключевого слова 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; }
}
}
```

Создадим объект класса-наследника, зададим значения сторон и вызовем метод вычисления площади

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

Если в классе-наследнике конструкторы описаны явно и требуется вызвать конструктор базового класса, используют ключевое слово base

```
public Rect():base()
```

При создании объекта сначала происходит вызов конструктора базового класса и только потом – конструктора класса-наследника.

Для обращения к скрытым членам в классе-наследнике также используют ключевое слово base, которое можно рассматривать как ссылку на базовый класс для класса, в котором base используется,

```
<base>.<член класса>
```

Полиморфизм

Все типы данных в языке C# относят к одной из двух категорий: типы значения и ссылочные типы. Объекты относятся к ссылочным типам данных *(reference type)*. Так как в языке 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`.

Таким образом, для использования свойства полиморфизма в языке программирования C# необходимо виртуальные методы перегрузить в классе-наследнике с ключевым словом `override`.

Лабораторная работа № 6
СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ КЛАССОВ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C#

Цель работы

Закрепление основ свойств объектно ориентированного программирования, изучение синтаксиса описания классов на языке программирования C#, описание пользовательского класса с заданными свойствами и методами.

Постановка задач

Описать на языке программирования C# класс Треугольник со следующими свойствами: две стороны и угол между ними, третья сторона должна быть вычислена на основе первых двух и угла между

ними. Описать два конструктора по умолчанию и с параметрами. Удаление объекта должно сопровождаться появлением надписи: «объект удалён». Перегрузить методы инкремента, декремента, оператор +, наделив их реальным геометрическим смыслом. Описать виртуальный метод вывода на экран данных об объекте. Описать класс Пирамида – наследник класса Треугольник. В основании пирамиды лежит треугольник родительского класса. Переопределить в классе-наследнике перегруженные операторы. Для классов иерархии продемонстрировать работу всех описанных методов.

Вопросы для закрепления материала

1. Какое ключевое слово нужно указать перед началом описания класса?
2. Какие две категории типов данных существуют в языке программирования C#?
3. Возможно ли в языке программирования C# множественное наследование?
4. С каким модификатором доступа нужно объявить член в базовом классе, чтобы он был не доступен для внешнего использования, доступен для наследования?
5. Какое имя имеет конструктор класса?
6. Какое количество конструкторов можно описать в классе?
7. Какое количество деструкторов можно описать в классе?
8. В каком методе должны быть описаны действия, выполняемые при уничтожении объекта?
9. Какое ключевое слово применяется для перекрытия обычных (невиртуальных) методов и свойств класса-наследника?
10. Какое ключевое слово используют в заголовке виртуального метода при его переопределении в классе-наследнике?
11. Для чего используют ключевое слово `base` при вызове метода?

Задания для самостоятельной работы

Реализовать на языке программирования C# задания для самостоятельной работы, приведённые после первой, второй, третьей и четвёртой лабораторных работ.

Каждый класс в задачах, приведённых ниже, должен содержать конструктор по умолчанию и конструктор с параметрами. Продемонстрировать работу со всеми описанными методами.

1. Разработать класс, описывающий комплексное число, содержащий метод вывода комплексного числа на экран, перегруженные операторы для работы с комплексными числами, вычисления модуля и нормы комплексного числа.

2. Разработать класс Вектор, задаваемый координатами на плоскости; методы класса должны обеспечивать допустимые над векторами операции.

3. Описать класс Трёхмерный вектор – наследник класса Вектор из предыдущей задачи, переопределить операторы. Метод вывода вектора на экран в классе Вектор объявить виртуальным.

4. Описать иерархию классов: базовый класс – Квадрат, класс-наследник – Прямоугольник. Разработать методы перемещения фигуры на плоскости, масштабирования, вычисления площади и периметра.

5. Разработать класс многочленов от одной переменной, задаваемых степенью многочлена и набором коэффициентов. Класс должен содержать методы вычисления значения многочлена для заданного аргумента, сложения, вычитания и умножения на число.

6. Описать класс Матрица. Предусмотреть возможность изменения количества строк и столбцов уже существующего объекта, умножения матрицы на число, сложения матриц, вывода на экран всей матрицы и её части заданного размера.

7. Разработать класс Записная книжка с не менее чем семью характеристиками абонента, возможностью работы с произвольным числом записей, поиска записи по значению какого-либо свойства, например по фамилии, дате рождения и т. д.

8. Описать класс Студенческая группа. Организовать возможность добавления и удаления записей, поиска по значению ключевого поля, вывода информации на экран.

9. Описать класс Домашняя библиотека. Предусмотреть возможность работы с произвольным числом книг, добавления и удаления книги, сортировки по заданной характеристике, поиска по какому-либо ключевому полю, вычисления суммарного количества страниц.

10. Описать иерархию классов: базовый класс Человек, классы-наследники Ребёнок, Ученик, Взрослый. Создать виртуальный метод вывода информации об объекте на экран.

Разработка визуального приложения на языке C#. Работа с формой

Теоретические сведения и методические указания к работе

Следующие разделы курса посвящены разработке визуальных приложений Windows Forms. После создания такого приложения решение будет содержать два модуля `Program.cs` и `Form.cs`. Структура проекта представлена на рис. 14.

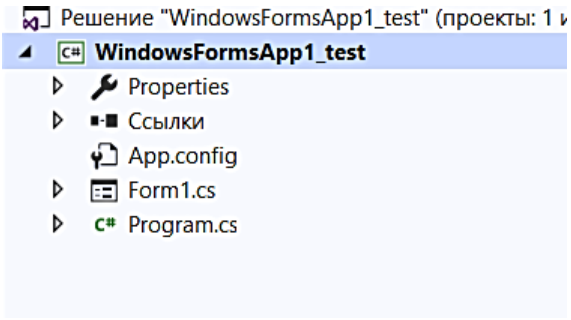


Рис. 14. Структура созданного проекта

Файл `Program.cs` содержит метод `Main()`, вызываемый 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 будет использована одиночная модель потоков для приложения. Этот атрибут должен быть указан для всех точек входа в Windows Forms-приложения.

Метод `Main()` содержит обращение к классу *Application*, включающему в себя статические свойства и методы поддержки работы приложения. Так как эти свойства и методы статические, для их вызова не нужно создавать объект.

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

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

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 добавлять и расставлять на форме необходимые компоненты.

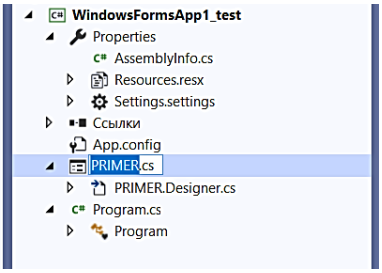
Именование формы

Все идентификаторы должны иметь уникальные имена, несущие смысловую нагрузку. Объект класса `Form` не исключение, поэтому работу с проектом начинаем с задания смыслового имени формы. По умолчанию созданная форма будет иметь имя `Form1`, удаляем единицу и добавляем какую-то смысловую часть. Предпочтительнее, чтобы это было одно слово, емко описывающее назначение приложения.

Для получения возможности редактировать имя нужно дважды щёлкнуть правой кнопкой мыши по имени 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 проделанных действий представлен на рис. 15.



а)

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

б)

Рис. 15. Изменение имени формы: а – выбор файла; б – результат изменений

Изменение свойств формы

Свойства формы и размещённых на ней компонентов можно изменять двумя способами: программно, т. е. в коде, и задавая соответствующие значения в окне редактора свойств.

Первый способ используют, если изменения должны произойти во время работы программы. Второй подход позволяет задать свойства компонентов во время разработки приложения; именно с этими начальными настройками компоненты отобразятся на форме при запуске приложения.

Текст, который отображён в верхней строке формы, хранится в строковом формате свойства `Text` формы. По умолчанию это значение совпадает с именем формы.

На рис. 16 показаны задание значения свойства и результат его изменения.

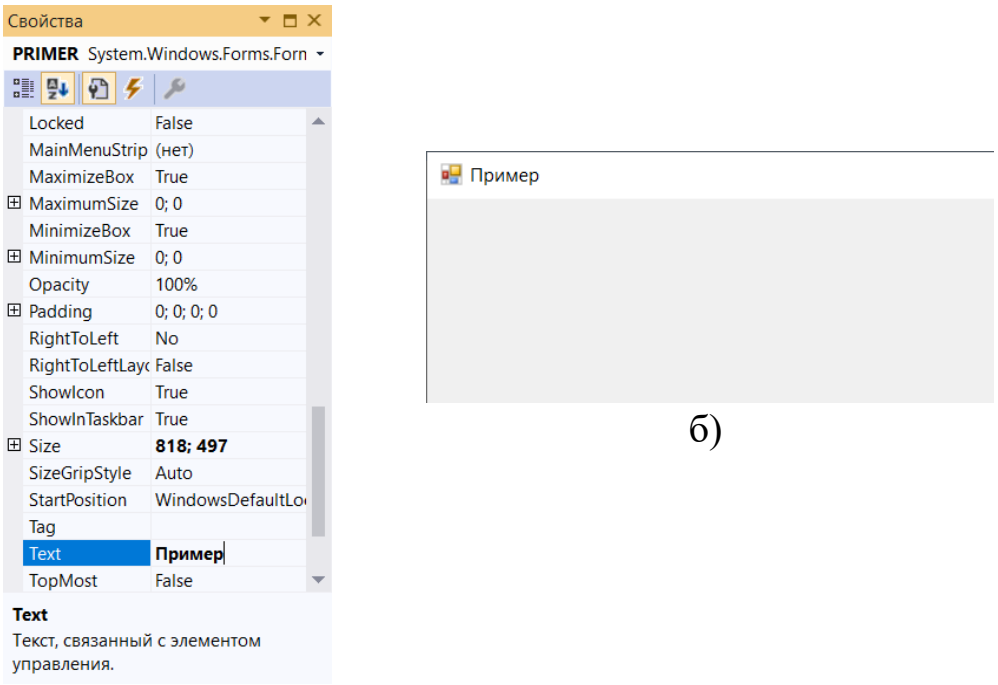


Рис. 16. Изменение значения свойства Text:
а – процесс изменения свойства; б – результат изменения свойства

Компонент `Label` предназначен для отображения на форме текста, обычно это сопроводительные надписи к другим компонентам. Разместим на форме этот компонент (рис. 17) и изменим надпись на

нём. По умолчанию при добавлении компонента на форму, если компонент имеет свойство `Text`, оно содержит имя объекта – название компонента и его порядковый номер. Таким образом, после добавления метки на форму можно увидеть текст «*label1*». Изменим это свойство программно, внесем изменения в конструктор формы модуля `PRIMER.cs`.

а) б)

Рис. 17. Добавление компонента на форму:
а – выбор компонента; б – результат добавления

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

При запуске формы на экране в месте, где размещена метка, появится соответствующая надпись. На рис. 18 показан результат запуска приложения.

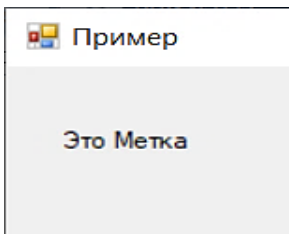
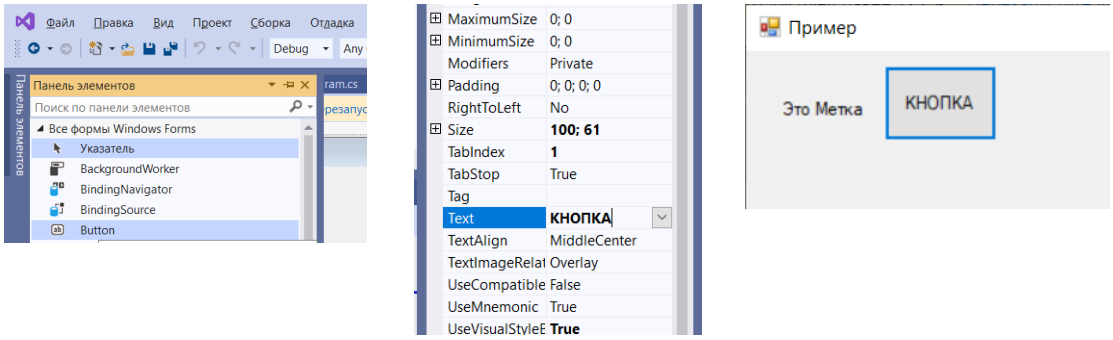


Рис. 18. Изменение текста метки

Добавим на форму кнопку – компонент `Button` и напишем обработчик события нажатия на эту кнопку, предварительно изменив надпись на ней (рис. 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 компоненту `Button` в режиме проектирования добавляет в код обработчик основного события компонента, для кнопки – нажатие на нее

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

Код, написанный в фигурных скобках, т. е. тело обработчика, выполнится при наступлении события, т. е. при нажатии на кнопку.

Компоненты. Свойства. Методы. События

Разработка визуального приложения

Разработка визуального приложения предполагает размещение на форме компонентов и написание обработчиков событий.

Все компоненты относятся к одной из двух групп: визуальные и невизуальные. Визуальные компоненты во время проектирования отображаются на форме так же, как и во время работы приложения. Невизуальные компоненты отображаются во время проектирования в виде пиктограмм; во время исполнения программы не видны до момента работы с ними.

Внешний вид и некоторые моменты поведения задаются свойствами компонента, например цвет, размер, возможность изменять значение некоторых полей во время работы приложения.

Как уже было упомянуто ранее, значение свойств компонента можно изменить после его размещения на форме. Это можно сделать двумя способами:

- изменить значение в окне редактирования свойств;
- программно, т. е. добавить строки кода, в результате выполнения которых во время работы программы произойдёт необходимое изменение значения нужного свойства.

Для программного изменения значения свойства какого-либо объекта необходимо в обработчике соответствующего события использовать оператор присваивания, указав в его левой части имя компонента, затем точку и имя свойства, значение которого планируется изменить:

<имя объекта>.<имя свойства>

В правой части оператора присваивания указываем необходимое значение.

Пример.

```
label1.Text= “метка”;
```

В том же окне есть закладка *События*.

Каждый компонент как объект своего класса имеет ряд событий. Событие происходит во время выполнения программы, например нажатие клавиши на клавиатуре или щелчок кнопкой мыши.

Если написать код обработчика события, то после наступления этого события управление будет передано первому оператору обработчика.

Каждый компонент имеет основное событие, то, ради обработки которого он был создан. Например, для кнопки `Button` это событие `Click`.

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

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

Разберём назначение некоторых основных свойств. Все свойства активного компонента можно найти в окне *Properties* (*Свойства*). Имя объекта доступно для редактирования и содержится в свойстве `Name`. По умолчанию объект получает имя, состоящее из двух частей: имени компонента со строчной буквы и порядкового номера компонента соответствующего типа, добавленного на форму. Например, `label1, label2.`

Всем объектам стоит давать осмысленные имена. При этом первая часть остаётся неизменной или сокращается без потери смысла и однозначности, порядковый номер заменяют на информативную часть, говорящую о роли объекта в приложении.

Например, `labelName` или `labName.`

Метка с таким именем может располагаться перед окном для ввода имени и содержать подсказку для пользователя или, наоборот, отображать на форме сведения, полученные каким-то другим способом, например прочитанные из файла.

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

Ряд свойств имеют свой набор свойств, которые можно увидеть в списке, раскрывающемся при выборе такого свойства. Например, свойство `Font` является самостоятельным классом со своими свойствами.

Работа с формой

Далее перечислены основные свойства формы, многие компоненты обладают такими же свойствами, поэтому имеет смысл эти свойства разобрать.

Имя формы, по которому к ней можно обращаться в коде, `Name`.

Заголовок окна содержится в свойстве `Text`.

Одно из трёх состояний окна (Normal, Maximized, Minimized) можно выбрать в свойстве WindowsState.

Цвет фона окна – BackColor.

Изображение на форме – BackgroundImage.

Выбрать вид отображения курсора при наведении на окно – Cursor.

Свойства шрифта, которым отображён цвет поверх окна, – Font.

Цвет переднего плана, цвет текста – ForeColor.

Иконка, отображаемая в верхнем левом углу формы, – Icon.

Отображение иконки в заголовке окна – ShowIcon, свойство имеет логическое значение.

Opacity в процентах по умолчанию установлено значение 100 %, т. е. окно абсолютно непрозрачно.

Размер формы, так же как и любого другого окна, можно изменить, перемещая его границы. При этом будут изменяться подсвойства Height и Width свойства Size.

Свойство Tag имеет тип Object, поэтому в нём можно сохранить любую информацию.

Методы формы

Работу приложения, в том числе и взаимодействие с пользователем, можно организовать, написав обработчики событий формы. Рассмотрим наиболее часто используемые из них.

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

Скрыть окно, не уничтожая его, – Hide().

Закрыть окно и уничтожить объект формы – Close().

Перерисовать форму – Invalidate().

Компоненты общего назначения и компоненты управления

Все визуальные компоненты, т. е. компоненты, которые отображаются на форме во время проектирования в том же виде, что и во время запуска программы, обладают свойствами, приведёнными ниже.

Name – имя компонента, используемое в коде.

Anchor – задаёт сторону формы, к которой прикреплён компонент.

BackColor – цвет фона компонента.

Cursor – задание вида, который будет принимать курсор при наведении на объект.

Dock – выравнивание компонента на форме. Это свойство может принимать одно из следующих значений:

- Fill – компонент заполняет все свободное пространство формы;
- Bottom – компонент зафиксирован у нижнего края формы;
- Top - компонент зафиксирован у верхнего края формы;
- Left – компонент зафиксирован у левого края формы;
- Right – компонент зафиксирован у правого края формы;
- None – компонент не зафиксирован, расположен в месте, указанном в свойстве Location.

Enabled – определяет возможность доступа к компоненту во время работы приложения. Свойство имеет логическое значение. По умолчанию установлено значение true, при котором компонент доступен, при установке значения – false компонент становится недоступным для обращения и не реагирует на связанные с ним события, при этом компонент виден на форме, но отображён светло-серым цветом.

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.

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, по умолчанию установлено значение true, компонент виден, при изменении значения на false компонент становится полностью невидимым до изменения значения этого свойства на противоположное.

Рассмотрим некоторые наиболее часто используемые компоненты, разберём их основные события и методы.

Как уже было упомянуто, добавления в код обработчика основного события компонента можно добиться двойным щелчком по этому

компоненту во время проектирования. Для создания обработчика другого события нужно выбрать это событие из списка событий соответствующего компонента и выполнить двойной щелчок по нужной строчке. После чего в правом столбце списка событий напротив названия выбранного события будет отображено имя обработчика, а в коде будет сгенерировано название соответствующего метода и добавлены пустые строки, внутри которых нужно писать тело обработчика.

На рис. 20 представлен вид окна, содержащего список методов компонента `Button` и добавления обработчика `MouseMove`.

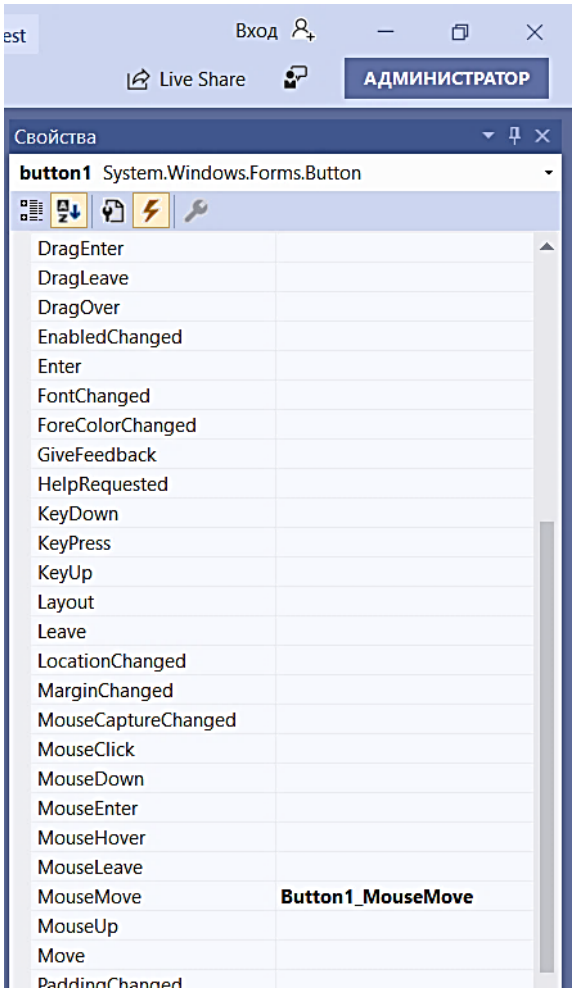


Рис. 20. Выбор события *MouseMove* для компонента *Button1*

В программу будет добавлен следующий код:

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

Назначение и основные свойства компонента Label

Этот компонент предназначен для отображения на поверхности формы текста, обычно это заголовок или какая-то поясняющая надпись.

Основные свойства.

`Text` – текст, отображаемый на поверхности компонента.

`AutoSize` – определяет размер компонента, по умолчанию имеет значение `true`, метка автоматически принимает размеры, минимально необходимые для отображения текста.

`TextAlign` – определяет сторону, по которой выровнен текст метки. Использование этого свойства имеет смысл при установлении свойства `AutoSize` в значение `false`.

`Font` – параметры шрифта.

Назначение, основные свойства и события компонента TextBox

Компонент ***TextBox*** позволяет пользователю вводить текст в специальное поле.

`Text` – это свойство имеет строковый тип, по умолчанию содержит пустую строку.

`Multiline` – разрешает или запрещает многострочное отображение данных.

`Modified` – имеет значение `true`, если пользователь внёс изменения в строку, изменить значение этого свойства на `false` можно только в коде.

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

`TextChanged` – событие, генерируемое при изменении пользователем текста в окне компонента.

Назначение, основные свойства и события компонента ComboBox

ComboBox – это раскрывающийся список. Этот компонент удобен при необходимости организовать выбор пользователем одного значения из списка.

`Items` – коллекция строк – элементов списка.

MaxDropDownItems – количество строк, видимых при раскрытии списка.

SelectedIndex – свойство, возвращающее номер выделенной строки, ни одна строка не выделена, значение свойства равно –1.

SelectedItem – текст выделенной строки, если ничего не выделено, то значение свойства null.

Перед использованием номера выделенной строки для предотвращения возникновения ошибок значение свойства необходимо проверить на null.

Пример.

```
if (comboBox1.SelectedItem !=null)
{string str = comxboBox1.SelectedItem.ToString();
MessageBox.Show(str);
}
else
MessageBox.Show(“ничего не выделено”);
```

Add() – метод добавления строки в коллекцию.

AddRange() – метод, принимающий массив строк и добавляющий эти строки в коллекцию элементов компонента.

Remove() – удаление строки из коллекции.

Clear – очистка списка коллекции.

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

SelectedIndexChange() – событие, возникающее при изменении индекса выделенной строки.

Назначение, основные свойства и события компонента CheckBox

Этот компонент позволяет пользователю выбрать одно из двух состояний. Используется, когда на какой-то вопрос нужно дать ответ: «да» или «нет».

Основные свойства.

Text – текст, поясняющий значение флажка.

Checked – имеет значение true, если флажок установлен, в противном случае – значение false.

AutoCheck – по умолчанию имеет значение true, при этом значение свойства checked изменяется при щелчке на изображении флажка.

TextAllign – выравнивание текста в компоненте.

CheckAllign – положение кнопки в компоненте.

События.

Click() – генерируется при щелчке по компоненту.

CheckedChange() – генерируется при изменении свойства Checked.

Назначение и основные свойства компонента CheckedListBox

Компонент ***CheckedListBox*** представляет собой список с индикаторами, элементы которого хранятся в свойстве Items.

Свойства.

Items – коллекция строк списка.

MultiColumn – при установлении значения true строки будут отображены в несколько колонок.

SelectedItem – содержит выделенный элемент списка, не выбранный, отмеченный галочкой, а именно выделенный, если ничего не выделено, свойство равно null.

Методы.

Add() – добавление строк в коллекцию;

Remove() – удаление строки из коллекции;

Clear() – очистка коллекции строк;

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

События.

SelectedIndexChanged – наступает при изменении индекса выделенного элемента.

SelectedValueChanged – наступает при изменении выделенного значения.

Назначение и основные свойства компонента ListBox

Этот компонент похож на предыдущий, но без флажков-индикаторов.

Свойства.

SelectedItem – значение текущего выбранного элемента.

SelectionMode – разрешение или запрет множественного выбора строк.

SelectedItem – содержит выделенный элемент списка.

SelectedItems – содержит все выделенные элементы списка при разрешённом множественном выборе.

Пример перебора выделенных в компоненте listBox строк и вывод их в отдельное окно.

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

События.

SelectedIndexChanged – наступает при изменении индекса выделенного элемента.

SelectedValueChanged – наступает при изменении выделенного значения.

Компонент DateTimePicker позволяет пользователю выбрать дату и время.

Основное свойство.

Value – хранит выбранную пользователем дату, имеет тип DateTime.

Основное событие.

ValueChanged – наступает при изменении даты или времени внутри компонента.

Компонент PictureBox создан для отображения картинки на форме.

Основное свойство компонента – Image.

SizeMode – режим отображения выбранной картинки, возможна установка одного из следующих режимов:

StretchImage – растягивает картинку поверх всей поверхности компонента;

AutoSize – компонент принимает размеры картинки;

CenterImage – отцентрировать картинку внутри компонента;

Zoom – уменьшение картинки так, чтобы она вписалась в компонент с сохранением пропорций изображения.

Normal – отображение картинки без масштабирования.

Компонент ProgressBar отображает процент выполнения каких-либо действий.

Свойства.

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

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

`Value` – текущее значение.

Компонент RadioButton – переключатель позволяет пользователю выбрать одно из двух состояний.

При размещении нескольких таких компонентов в контейнере можно выбрать только один из них. Под контейнером понимают любой компонент, на поверхности которого можно разместить другие компоненты.

Свойства.

`Text` – строка, содержащая текст, отображаемый около индикатора.

`Checked` – имеет значение `true`, если компонент выбран.

`AutoCheck` – при значении `true` значение свойства `Checked` становится равным `true` при щелчке по компоненту, в противном случае, если значение свойства `AutoCheck - false`, при щелчке по этому компоненту он автоматически не выделится. Выделение можно сделать в обработчике события `Click` на этом компоненте.

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

Основное событие `Click` – срабатывает при щелчке по кнопке во время 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».

Свойства.

`Checked` – имеет значение `true,` если значение выбрано, т. е. в компоненте стоит флажок, иначе свойство имеет значение `false`.

`CheckState` – состояние позволяет выбрать одно из трёх состояний компонента, в котором он может находиться: `Checked` – отмечен, `Indeterminate` не определен – отмечен, но находится в неактивном состоянии и `Unchecked` – не отмечен.

События.

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

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

`Click` – срабатывает при щелчке по компоненту.

Компоненты контейнеры

Контейнерами называют компоненты, на поверхности которых можно размещать другие объекты. К их помощи прибегают для визуального объединения компонентов, предназначенных для решения общей задачи.

Group Box

Часто используют для группировки нескольких компонентов `RadioButton`, в этом случае 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.

Panel

Для смысловой группировки компонентов на форме обычно добавляют несколько панелей, разделяя их компонентом `Splitter`. Работу с панелями начинают с задания свойства `Dock`, размещающего панель вдоль какого-либо края формы.

TabControl

Набор страниц с возможностью переключения между ними. Каждая страница содержит заголовок.

Свойства.

`Text` – содержит заголовок страницы.

`ToolTipText` – содержит текст подсказки для страницы. Подсказка отображается при наведении курсора на заголовок страницы.

`ShowToolTips` – необходимо установить в значение `true`, чтобы текст подсказки, содержащейся в свойстве `ToolTipText`, отображался при наведении курсора на страницу.

`TabPages` – коллекция страниц.

`TabIndex` – индекс текущей страницы.

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

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

Лабораторная работа № 7.
РАЗРАБОТКА ВИЗУАЛЬНОГО ПРИЛОЖЕНИЯ

Цель работы

Совершенствование навыков объектно ориентированного программирования. Знакомство с визуальными компонентами. Получение навыков разработки простейших обработчиков событий.

Постановка задачи

Создать *Windows*-приложение для решения следующих задач.

1. Задать треугольник, используя один из этих трёх способов: через основание и высоту; через две стороны и угол между ними; через три стороны. При выборе пользователем способа задачи треугольника активировать компоненты, позволяющие ввести необходимые данные и вычислить площадь треугольника.

2. Спортсмен начинает тренировки и пробегает в первый день S км. Каждый следующий день он увеличивает дневную норму на 10 % по сравнению с нормой предыдущего дня. Какой суммарный путь пробежит спортсмен за N дней? Параметры S и N задаёт пользователь в соответствующих окнах ввода.

3. Амеба каждые три часа делится на две клетки. Определить, сколько амеб будет через N часов. Начальное количество амеб и необходимое число часов задать входными параметрами.

Указания к работе

Проект оформить в виде одного оконного приложения, содержащего компонент `TabControl`. Разместить решение каждой задачи на отдельной странице, снабдив ее всплывающими подсказками. Формулировку каждой задачи отобразить на соответствующей странице.

Для ввода данных использовать компоненты `TextBox`, для сопровождающих надписей – компоненты `Label`, вычисления производить в обработчике события нажатием на кнопку компонента `Button`, расположенного на соответствующей странице. Для всех значений, отображающихся на форме, должны быть указаны единицы измерения.

Вопросы для закрепления материала

1. Каким образом можно изменить значение свойства компонента?
2. Для каких целей используют компонент *label*?
3. Что содержит свойство *Text*?
4. Для каких целей используют компоненты контейнеры?
5. Перечислите известные вам компоненты контейнеры.
6. Какие действия необходимо произвести для добавления в программу автоматически сгенерированного кода обработчика основного события компонента?
7. Какие компоненты следует использовать для получения от пользователя утверждения: «да» или «нет»?
8. Для каких целей используют компонент *Button*?
9. Какое основное событие компонента *Button*?
10. Какие компоненты используют для организации выбора пользователем одного или нескольких значений из списка?

Задания для самостоятельной работы

Во всех задачах, представленных ниже, необходимо организовать проверку на корректность значений, вводимых пользователем. Для всех значений, вводимых пользователем, и результатов вычислений должны быть отображены соответствующие единицы измерения.

1. Палка длиной x м стоит около стены под наклоном. Один ее конец находится на расстоянии y м от этой стены. Вычислить значение угла α между палкой и полом.
2. Вывести на форму всевозможные сочетания количества гусей и кроликов на ферме, если известно, что вместе у них 64 лапы.
3. Вычислить сумму вклада на счёте, учитывая процентную ставку. Сумма вклада на год рассчитывается так:

Сумма = Сумма · Ставка/100.

Сумма на *n* лет рассчитывается следующим образом:

Сумма = $\sum_{i=0}^{n}$ Сумма · Ставка/100.

4. Вычислить стоимость поездки на автомобиле. Пользователь должен ввести следующие данные: цена одного литра бензина, расход бензина на 100 км, пройденное расстояние.

5. Вычислить сопротивления резисторов, соединенных параллельно или последовательно. Пользователь выбирает вид соединения и сопротивление одного резистора. Выбор соединения организовать, используя компонент *RadioButtonList*. Для ввода данных использовать компоненты *TextBox*.

6. Создать приложение «Конвертер», позволяющее переводить единицы измерения длины. Организовать ввод пользователем значения, выбор начальных и итоговых единиц измерения. Входные данные: расчет производить в обработчике события *Click* компонента *Button*.

7. Написать программу для решения квадратного уравнения вида $Ax^2 + Bx + C = 0$. Организовать ввод пользователем коэффициентов и возможность отображения справочной информации о решении по запросу пользователя.

8. Создать приложение «Секундомер». Организовать отображение текущего времени, интервала смены показаний текущего времени, кнопку для запуска и остановки секундомера, а также сброса текущего времени. Для управления секундомером использовать стандартный невизуальный компонент *Timer.*

9. Разработать приложение для создания списка преподавателей и студентов. Организовать возможность просмотра списка и добавления в него. Предусмотреть поля ввода данных: фамилии, группы или кафедры, а также выбор статуса – студент или преподаватель, кнопки для внесения в список и отображения списка студентов или преподавателей. Для решения задачи описать базовый класс *Person* и два класса наследника *Teacher* и *Student*. Для хранения объектов этих классов использовать массив базового класса.

10. Создать приложение «Расписание». По запросу пользователя на форме отображается расписание на указанный день.

11. Создать приложение «Восточный календарь». Отобразить название животного, символизирующего год по восточному календарю. Год вводит пользователь.

Лабораторная работа № 8.
ОРГАНИЗАЦИЯ ВЫБОРА ПОЛЬЗОВАТЕЛЕМ НУЖНОГО ВАРИАНТА ИЗ СПИСКА

Цель работы

Совершенствование навыка работы в визуальной среде. Углубление изучения свойств и методов базовых компонентов.

Постановка задачи

Разработать приложение «Меню».

Предложить пользователю выбор блюд из списка с указанием их цены и количества около выбранной позиции с возможностью добавления и уменьшения количества порций выбранной позиции. Рассчитать суммарную стоимость заказа.

Указания к работе

Выбор списка блюд и их цены организовать при помощи компонента *CheckListBox*.

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

При выборе пользователем строки меню или при отмене выбора стоимость заказа должна быть пересчитана так же, как и при изменении количества заказа выбранной позиции.

Вопросы для закрепления материала

1. С помощью каких компонентов можно организовать выбор пользователем одного или нескольких вариантов из списка?
2. Используя какой метод можно добавить строку в коллекцию?
3. Используя какой метод можно добавить сразу несколько строк в коллекцию?
4. Какой метод необходимо использовать для удаления строки из коллекции?
5. Какой метод необходимо использовать для очистки списка строк коллекции?
6. Какие действия нужно произвести для перебора всех строк коллекции компонента *CheckListBox*?

7. Какие действия нужно произвести для перебора всех выбранных строк компонента *CheckListBox*?

8. Какие действия нужно произвести для проверки статуса выбора определенной строки компонента *CheckListBox?*

9. Какие действия нужно произвести для получения значения выделенной строки компонента *CheckListBox?*

10. Какие действия нужно произвести для получения номера выделенной строки компонента *CheckListBox?*

Задания для самостоятельной работы

1. Создать приложение, позволяющее пользователю осуществить выбор из предложенного списка одного или нескольких иностранных языков для изучения. Выбранные языки отобразить в отдельном окне.

2. Создать приложение, позволяющее скопировать выбранные в компоненте *CheckListBox* строки в текстовое поле любого предназначенного для отображения текста компонента.

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

4. Добавить на форму кнопку и написать обработчик события нажатия на нее, добавляющий в компонент *CheckListBox* новую строку, введённую в текстовое поле компонента *TextBox.*

5. Добавить на форму компоненты *CheckListBox* и *ListBox*, написать обработчик события нажатия на кнопку для добавления в алфавитном порядке строк, выбранных в компоненте *CheckListBox,* в компонент *ListBox.*

6. Написать программу, позволяющую выбрать из списка с флажками несколько строк, и отобразить их на форме, удалив при этом выбранные строки из списка.

7. Написать обработчик события *SelectedIndexChanged* компонента *CheckListBox*, позволяющий сохранить текст всех выделенных строк в файле.

8. Создать визуальное приложение, позволяющее организовать при помощи компонента *CheckListBox* выбор пользователем предметов для дополнительного изучения, количество выбранных предметов должно быть не менее двух и не более четырёх.

9. Добавить в компонент *CheckListBox* строки из выбранного файла, свойство *Checked* первой добавленной строки задать значением *true,* остальных – *false*.

10. Организовать выбор пользователем видов досуговой деятельности при помощи компонента *CheckListBox*, выбранные значения сохранить в текстовом файле.

Лабораторная работа № 9
РАЗРАБОТКА ОБРАБОТЧИКОВ СОБЫТИЙ. КАЛЬКУЛЯТОР

Цель работы

Совершенствование навыков разработки обработчиков событий.

Постановка задачи

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

Указания к работе

Калькулятор должен обладать возможностью удаления последнего символа вводимого операнда.

При повторном нажатии на кнопку «равно» должно быть произведено выполнение последней выполненной операции с использованием последнего результата в качестве первого операнда, значение второго операнда остаётся прежним. Например, 4 + 3 = 7 = 10 = 13 = 16 и т. д.

При повторном нажатии на знак действия и сразу за ним на знак «равно» должна выполняться операция с использованием последнего результата в качестве первого и второго операндов. Например, 3 + 2 = 5 + = 10 - = 0.

Организовать вывод соответствующего сообщения в случае попытки деления на ноль.

Вопросы для закрепления материала

1. Какое событие называют основным для компонента?
2. Какое событие является основным для компонента `Button`*?*
3. Какие действия нужно произвести, чтобы добавить в код обработчик события, основного для компонента?
4. Каким образом можно, не дублируя код, организовать выполнение действий по одному алгоритму, но с разными параметрами при наступлении одного и того же события для разных компонентов одного класса?

5. Какие действия необходимо выполнить в коде для предотвращения ошибок пользовательского ввода?

6. Какие действия нужно выполнить в коде, чтобы отобразить на форме результаты вычислений?

7. Назовите основное событие компонента *CheckBox.*

8. Какое свойство и как нужно использовать, чтобы скрыть панель вместе со всеми расположенными на ней инструментами после завершения вычислений?

9. Когда наступает событие *CheckedChanged* компонента *RadioButton*?

10. Какой компонент предпочтительнее использовать для предоставления пользователю возможности задания целого числа из некоего диапазона?

Задания для самостоятельной работы

1. Создать приложение «Калькулятор комплексной арифметики». Калькулятор должен обладать следующим функционалом: ввод операндов, вычисление суммы двух комплексных чисел, разности, умножения, деления; результат вычислений сохранить в первом операнде и отобразить на форме.

2. Создать приложение «Калькулятор векторной алгебры». Калькулятор должен обладать следующим функционалом: ввод операндов, вычисление суммы двух векторов, разности, умножения, умножения на скаляр, результат вычислений, отображение результата вычислений.

3. Создать приложение «Тригонометрия», реализующее тренажёр для запоминания значений синуса и косинуса углов: 0°, 30°, 45°, 60°, 90°, 180°.

4. Создать приложение «Закон Ома для участка цепи». Реализовать выбор пользователем искомой величины и задания входных значений известных величин.

5. Создать приложение «Таблица умножения». Программа должна представлять собой тренажёр для запоминания значений таблицы умножения с возможностью выбора пользователем значения множителя или диапазона множителей для запоминания таблицы умножения и количества задаваемых примеров. Каждый введённый ответ должен быть сразу проанализирован с демонстрацией пользователю результата (правильно/неправильно). При завершении тестирования необходимо представить пользователю информацию о количестве правильных и неправильных ответов.

6. Создать приложение «Меню» с возможностью выбора категории блюд и выбора блюда из соответствующей категории с указанием количества по каждой позиции и расчётом полной стоимости заказа.

7. Создать приложение «Матрица» с возможностью задания пользователем размерности матрицы, ввода значений. Реализовать следующие операции: сложение, умножение на число и матрицы на матрицу, транспонирование.

8. Создать приложение «Конвертер» с возможностью выбора пользователем категории (масса, длина, информация, скорость, время и другие), ввода пользователем значений, выбора входных и итоговых единиц измерения.

9. Создать приложение «Электронные весы» с возможностью расчёта калорийности блюда по выбранным из списка ингредиентам и их весу.

10. Создать приложение «Расписание», позволяющее увидеть расписание и домашнее задание на день, введенный пользователем.

Работа с графикой

Теоретические сведения и методические указания к работе

`Graphics` – класс, находящийся в пространстве имен `System.Drawing`. Это основной класс для работы с графикой в *C#*.

Поверхность и методы рисования реализованы в классе Drawing. Для получения объекта этого класса используют обработчик события `Paint`.

Ниже представлен обработчик события `Paint` для формы

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

            }
```

Второй параметр метода – это переменная класса `PaintEventArgs`, через неё мы получаем доступ к области, которую необходимо перерисовать – `ClipRectangle`.

Экземпляр класса Graphics рассматривают как поверхность для рисования. Для отображения на ней графических объектов необходимо задать их координаты (передать в качестве параметров в соответствующие методы). Для корректного расчёта координат необходимо знать положение точки (0; 0) – это левый верхний угол поверхности.

Итак, для отображения на форме графических объектов в обработчике события Paint необходимо использовать объект e.Graphics и его методы. В обработчиках других событий объект класса Graphics нужно создавать самостоятельно. Для этого используют метод CreateGraphics формы или другого элемента управления, на котором необходимо отобразить графику.

Для рисования на форме код создания объекта класса Graphics будет таким:

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

Если необходимо отобразить объекты на заранее созданном контейнере, например панели, создание объекта класса Graphics будет выглядеть так:

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

Основные методы класса Graphics

Clear( ) – очистка поверхности рисования и заливка ее цветом, переданным в качестве параметра;

DrawLine( ) – отображение линии с указанными координатами начала и конца;

DrawRectangle( ) – отображение на форме прямоугольника;

DrawEllipse( ) – отображение на форме эллипса;

FillEllipse( ), FillRectangle( ) – отображение на форме соответствующих фигур и заливка их цветом.

DrawArc ( ) – отображение дуги.

Примеры использования основных методов класса Graphics

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

Вся поверхность для рисования будет залита красным цветом.

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

На поверхности будет отображена линия зелёного цвета толщиной 2 пикселя с координатами начальной точки (10; 10) и конечной точки (100; 100).

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

На поверхности будут отображёны прямоугольник линией толщиной в 3 пикселя синего цвета, координаты левого верхнего угла прямоугольника (10; 10), правого нижнего угла (100; 100).

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

На поверхности будет отображён эллипс черного цвета с координатами левого верхнего угла (15; 15) и правого нижнего угла (125; 125), толщина линии – 2 пикселя.

Для задания цвета фигур используют структуру типа `Color`. Структура имеет только конструктор по умолчанию. После создания структуры её поля изменить нельзя

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

Цвет можно задать при помощи статических методов *Color,* например метода `FromName`*()*, принимающего в качестве параметра название цвета в виде строки

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

Если необходим один из основных цветов, то используют предопределённые значения статических свойств структуры `Color`

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

Цвет и толщину линии фигуры задают через класс `pen`, заливку определяет класс `Brush`.

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

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

`DashStyle` – свойство объекта класса `pen`, задающее стиль отображения линии.

`Brush` – абстрактный класс, следовательно, нельзя создать экземпляр этого класса. Однако можно создать экземпляр класса-наследника класса `Brush`, например класса `SolidBrush` (сплошная заливка)

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

Создана кисть для сплошной заливки красным цветом.

Выберем в качестве поверхности для рисования панель, размещённую на форме, и приведём простой пример отображения на ней графических объектов

```
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)
}
```

Работа всех строк кода примера была разобрана выше. Результат работы приведённого фрагмента кода представлен на рис. 21.

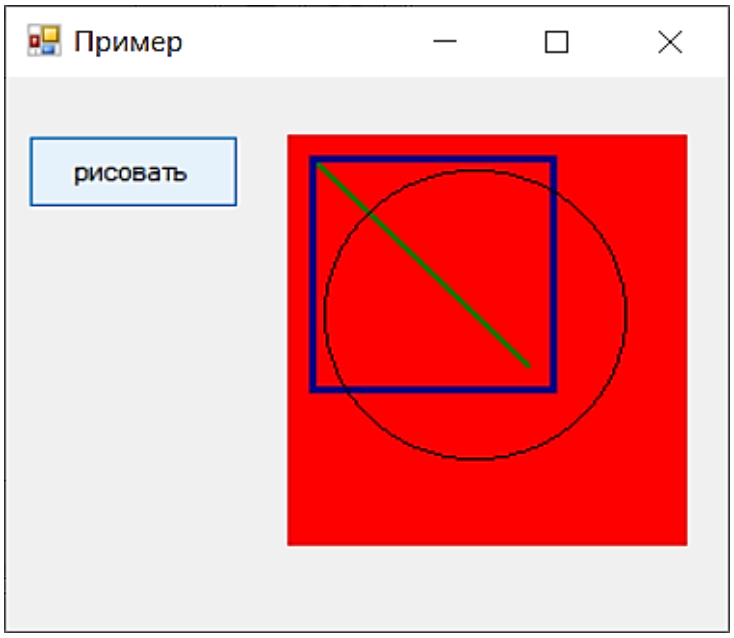


Рис. 21. Результат работы кода обработчика события нажатия на кнопку «Рисовать»

Для работы с изображениями, сохранёнными в файлах, используют объекты класса `Image`.

Основные свойства класса Image

`Size` – размер картинки в пикселях;
`Width` – ширина картинки;
`Height` – высота картинки;

`PixelFormat` – формат пикселя;
`RawFormat` – формат файла изображения.

Основные методы класса Image

`Clone( )` – возвращает копию объекта;
`GetBounds( )` – возвращает размеры картинки;
`RotateFlip( )` – поворот изображения;
`Save( )` – сохранение картинки в файле;
`FromFile( )` – создание экземпляра класса `Image`, загрузка в него картинки, возврат объекта в точку вызова.

Создадим объект класса `Image` и загрузим в него изображения из файла:

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

Загрузку картинки из файла рекомендуется делать в блоке исключений `try` для предотвращения прерывания работы программы из-за ошибки загрузки. При некорректном формате файла может возникнуть ошибка `OutOfMemoryException`.

Далее приведен код обработчика нажатия на кнопку, позволяющий загрузить изображение из файла и отобразить его на панели.

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

На рис. 22 приведен результат работы обработчика.

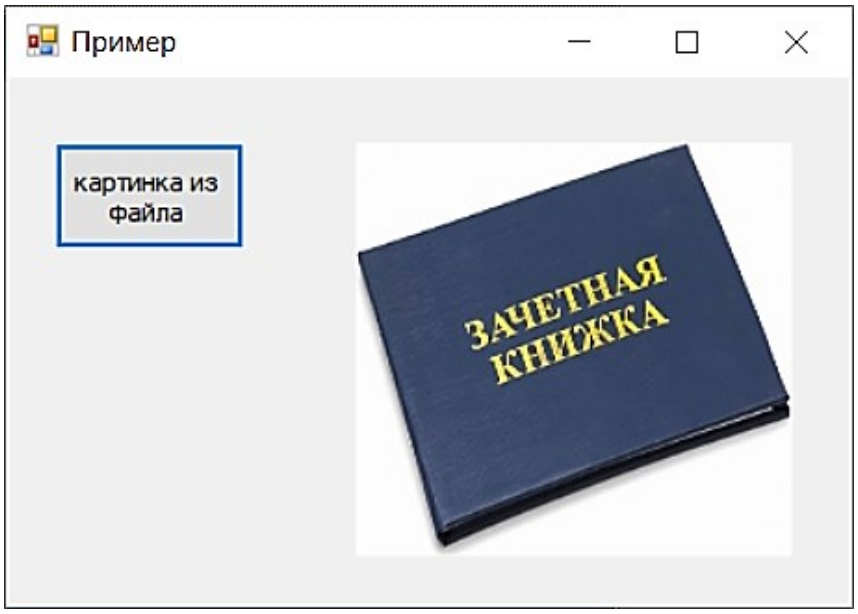


Рис. 22. Отображение картинки, загруженной из файла

В примере, приведённом выше, после создания объекта класса `Image` происходит работа с компонентом класса `OpenFileDialog`. Это не визуальный компонент, т. е. он не виден на форме во время работы приложения до определённого момента. Создать компонент можно как во время проектирования, так и во время работы программы.

Создадим объект программно и зададим его свойство `Filter`. Работа обработчика будет прервана, если имя файла не будет выбрано. Если имя файла выбрано, работа обработчика продолжается и при удачном открытии файла изображение передаётся в объект `image`; если имя файла не выбрано, работа обработчика прервётся.

Если до этого момента всё прошло успешно, будет создан объект класса `Graphics` и вызван метод `DrawImage`, отображающий объект `image` в заданных координатах. Так как это координаты левого верхнего и правого нижнего углов, изображение займёт всю поверхность панели.

Лабораторная работа № 10
ОТОБРАЖЕНИЕ ПРОСТЕЙШИХ ГРАФИЧЕСКИХ ОБЪЕКТОВ

Цель работы

Знакомство с принципами отображения графики в языке программирования C#.

Постановка задачи

Разработать визуальное приложение, позволяющее отображать на форме или любой другой поверхности предложенные пользователю на выбор графические объекты.

Для этого на форме разместить компоненты, позволяющие организовать выбор не менее чем из трёх видов рисунков (например, звезда, снеговик, дом), задание цвета и толщины линии.

Добавить на форму две кнопки; при нажатии на одну из них выбранный объект будет отображён, на другую – удален.

Вопросы для закрепления материала

1. Какой класс используют для работы с графикой в языке программирования C#?
2. В каком пространстве имён находится этот класс?
3. Какие основные свойства этого класса?
4. Какие методы этого класса вы знаете?
5. Каким образом можно задать цвет для отображения фигуры?
6. Какое свойство нужно использовать для задания стиля отображения линии?
7. Что принято за начало координат поверхности для отображения графики?

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. В предыдущее задание добавить настройку цвета отображения фигуры.

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

4. В предыдущее задание добавить регулятор скорости перемещения фигуры по траектории.

5. В предыдущее задание добавить регуляторы диапазона и скорости масштабирования.

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

7. Разработать приложение «Графический редактор», обладающее 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 отпущена. Использовать 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. Нажатие в этом состоянии на левую клавишу мыши изменяет форму курсора. Если курсор находился на конце линии, то его форма 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.

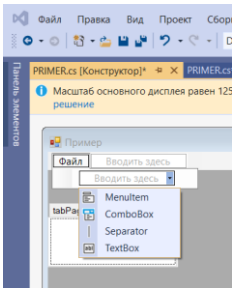
10. Разработать визуальное приложение. При запуске программы на форме приложения появляется геометрическая фигура, которая будет рассмотрена как траектория движения другой фигуры. Предусмотреть: регулятор изменения размера (масштабирования) траектории, настройку цвета фона рабочей области, цвета прорисовки 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

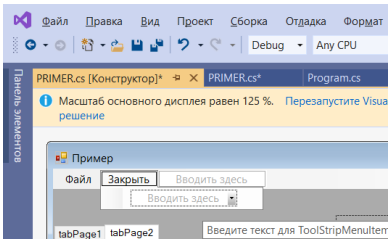
Для организации главного меню приложения используют компонент `MenuStrip`. При его добавлении на форму под строкой заголовка появляется панель меню. По умолчанию имя этого компонента `menuStrip1`. Свойство `MainMenuStrip` формы будет содержать это имя.

Для добавления разделов меню необходимо щелкнуть по прямоугольнику с надписью: «Вводить здесь» и ввести нужный текст. Двойной щелчок по пункту меню создаёт для него обработчик события `Click`. При создании подпунктов меню предлагается выбор типа подпункта, это может быть элемент меню или выпадающий список, поле для ввода или разделитель.

Процесс создания пунктов и подпунктов меню показан на рис. 23.



а)



б)

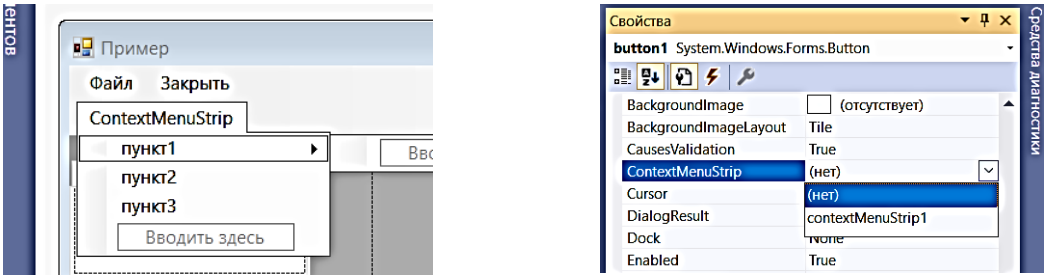
Рис. 23. Создание пунктов и подпунктов меню:
а – создание пунктов; б – создание подпунктов

Для завершения работы приложения и закрытия формы необходимо использовать метод `Close()`. Это можно сделать в соответствующем пункте меню.

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

Для создания контекстного меню какого-либо компонента, размещённого на форме, используют невизуальный компонент `ContextMenuStrip`, представленный на рис. 24. Такое меню становится видно при щелчке правой кнопкой мыши по элементу управления или форме. В меню можно добавлять только подпункты, разделы добавлять нельзя.

После добавления компонента на форму и задания нужных подпунктов меню нужно связать с тем компонентом, для которого оно создавалось. Выбираем нужный элемент управления в списке компонентов, добавленных на форму, в выпадающем списке свойства `ContextMenuStrip` выбираем соответствующее меню.



а) б)

Рис. 24. Компонент `ContextMenuStrip`:
а – задание пунктов меню; б – связь с компонентом `button1`

При необходимости создания на форме панели инструментов используют компонент `ToolStrip`. При добавлении на форму компонент будет расположен по её правому краю. На этой панели возможно размещение следующих компонентов: кнопок, меток, кнопок с всплывающим меню, разделителей, текстовых полей, индикаторов прогресса, выпадающих списков. На рис. 25 показано окно добавления инструментов.

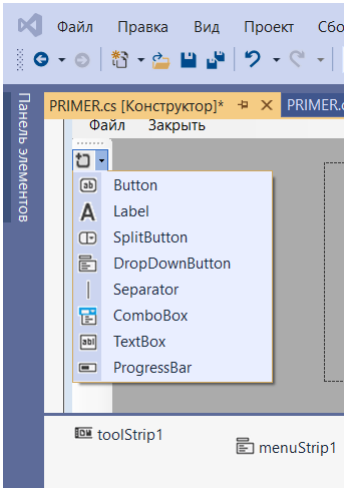


Рис. 25. Окно добавления
инструментов на панель

Для информирования пользователя о состоянии работы программы можно использовать компонент `StatusStrip`. На его поверхности можно разместить компоненты для отображения информации о процессе работы. Выбор соответствующих элементов представлен на рис. 26.

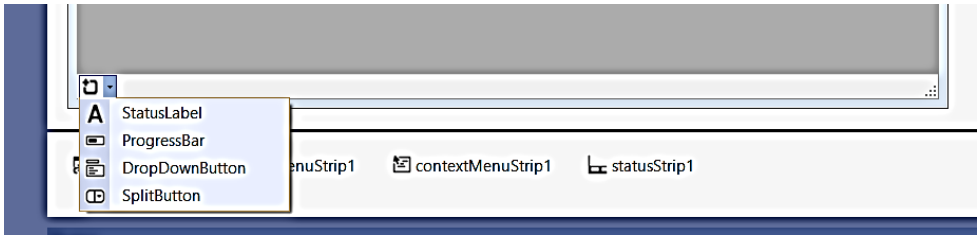


Рис. 26. Добавление элементов в компонент `StatusStrip`

Диалоги

Теоретические сведения и методические указания к работе

Напомним, что в библиотеке *.NET Framework* есть два типа компонентов: визуальные и невизуальные.

Визуальные компоненты – это элементы пользовательского интерфейса, например: кнопка (`Button`), выпадающий список (`comboBox`), метка (`Label`) и др.

Невизуальные компоненты не имеют пользовательского интерфейса. Во время проектирования, если такие компоненты были добавлены, их можно увидеть внизу окна дизайнера. Невизуальными компонентами являются таймер (Timer), компоненты для работы с базами данных, контекстное меню, различные диалоговые окна.

Компонент SaveFileDialog предназначен для организации диалога с пользователем и дальнейшего сохранения информации в файле с указанным именем. Компонент расположен на вкладке Dialogs панели Toolbox.

Основные свойства и методы компонента SaveFileDialog

InitialDirectory – содержит начальную папку для сохранения файла.

FileName – имя файла, выбранного в диалоговом окне.

Filter – строка фильтра, задающая варианты, появляющиеся в окне «сохранить как тип файла» или «файлы типа» в диалоговом окне.

FilterIndex – индекс фильтра, выбранного в настоящий момент в диалоговом окне файла.

Title - заголовок рассматриваемого диалогового окна.

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 чтобы сохранить текст, например из компонента richTextBox, необходимо активизировать компонент SaveFileDialog и задать необходимые параметры.

Например, следующим образом:

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

Загрузить в компонент `richTextBox` текст из выбранного файла можно так:

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

Лабораторная работа № 11
РАБОТА С ФАЙЛАМИ, ДИАЛОГОВЫМИ ОКНАМИ, СОЗДАНИЕ ГЛАВНОГО И КОНТЕКСТНОГО МЕНЮ. ПРИЛОЖЕНИЕ «АНКЕТА»

Цель работы

Совершенствование навыков разработки обработчиков событий.

Знакомство с принципами организации в приложении главного и контекстного меню, сохранения и загрузки данных из файлов, работа с компонентами отображения текстовой информации.

Постановка задачи

Создать приложение «Анкета», позволяющее получить информацию о каком-либо объекте, через заполнение пользователем соответствующих полей. Организовать возможность сохранения этой информации в файле, чтения информации из файла и отображения в соответствующем окне формы.

Указания к выполнению работы

Разместить на форме два компонента `panel`, разделив их компонентом `splitter`.

Одну панель выровнить по левому краю и разместить на ней компоненты для сбора информации об объекте. Выровнить вторую панель по ширине и разместить на ней компонент `richTextBox` для отображения всей информации об объекте.

На левой панели обязательно должны быть размещены следующие компоненты: `TextBox`, `CheckedListBox`, `ComboBox`, `ListBox`, `CheckBox`, `DateTimePicker`. Возможно использование по желанию дополнительных компонентов.

Написать обработчик события `Click` компонента `Button`, расположенного на правой панели, реализовать в нём сбор информации об объекте; из компонентов, расположенных на левой панели, скопировать информацию в компонент `richTextBox`, расположенный на правой панели.

Панели и все размещенные на них объекты должны сопровождаться надписями, поясняющими их назначение. Для панелей и меток изменить значения по умолчанию свойств `Color` и `Font` и его подсвойств `Size`, `Style`, `Color`.

На правой панели разместить три кнопки. Нажатие на первую приводит к копированию информации из левой панели в окно `richTextBox`. Нажатие на вторую активирует диалоговое окно для сохранения информации в текстовый файл; на третью – выбирает файл, информация из которого будет загружена в окно `richTextBox`.

Продублировать перечисленные выше три действия пунктами главного меню.

Добавить контекстное меню для компонента `richTextBox`, содержащее пункты «открыть», «сохранить», «очистить». По желанию в меню можно реализовать дополнительные пункты.

Вопросы для самоконтроля

1. Для чего используют компоненты контейнеры?
2. Какие компоненты контейнеры вы знаете?
3. Каким образом можно отобразить заголовок контейнера?
4. Какие компоненты используют для организации выбора пользователем одного из двух вариантов?
5. Какие компоненты используют для организации выбора пользователем нескольких вариантов из списка?
6. Какие компоненты используют для организации выбора одного варианта из нескольких предложенных?
7. Какой компонент используют для организации выбора пользователем даты и времени, какие свойства этого компонента позволяют это сделать?

8. Какой компонент используют для организации главного меню приложения?

9. Какие действия нужно совершить для добавления нового пункта главного меню приложения?

10. Какое меню называют контекстным?

11. Каким образом можно создать контекстное меню для конкретного компонента?

12. Какие действия нужно выполнить пользователю для реализации всплывающих подсказок при наведении курсора на объект?

13. Чем отличаются визуальные компоненты от невизуальных?

Задания для самостоятельной работы

Во всех представленных ниже заданиях необходимо создать визуальное приложение на языке *C#*. Приложение должно позволять пользователю ввести все необходимые данные, при этом должны отслеживаться ситуации ошибок пользовательского ввода. Ошибка ввода не должна вызывать прерывание работы программы, необходимо вывести соответствующее сообщение и позволить осуществить повторный ввод. Реализовать сбор введённых данных и отображение их в отдельном окне с возможностью его редактирования, очистки, сохранения данных в файле и загрузки данных из выбранного файла.

1. Создать приложение «Домашняя библиотека» с возможностью просмотра информации обо всех имеющихся книгах и добавления новых.

2. Создать приложение «Картинная галерея» с возможностью просмотра информации обо всех имеющихся картинах и добавления описания новых.

3. Создать приложение «Зоопарк» с возможностью просмотра информации обо всех животных и добавления описания новых животных.

4. Создать приложение «Ветеринарная клиника» с возможностью просмотра информации обо всех пациентах и добавления данных новых пациентов.

5. Создать приложение «Магазин игрушек» с возможностью просмотра информации о существующих товарах и добавления данных новых товаров.

6. Создать приложение «Кафе» с возможностью просмотра информации о существующих блюдах и добавления описания новых блюд.

7. Создать приложение «Каталог косметики» с возможностью просмотра информации о существующих товарах и добавления новых.

8. Создать приложение «Красная книга Владимирской области» с возможностью просмотра информации о редких животных области и добавления новых исчезающих видов.

9. Создать приложение «Гербарий» с возможностью просмотра информации о существующих экземплярах растений и добавления описания новых.

10. Создать приложение «Филателист» с возможностью просмотра информации о существующих марках и их количестве в коллекции и добавления новых марок и их описания.

ПРИМЕРЫ ИТОГОВЫХ ЗАДАНИЙ

Во всех заданиях необходимо разработать визуальное приложение на языке программирования C#.

При выполнении итоговых заданий необходимо учесть следующее: все идентификаторы, использованные в коде, должны иметь информативные имена, т. е. из имени должен быть понятен тип и назначение объекта, не должно быть имён: *mas*, *ob1*, *label1*, *storona* и им подобных.

Разместить на форме:

– компоненты для ввода пользователем необходимых характеристик объекта (полей объекта: сторон, радиуса и других), сопроводив надписями с указанием единиц измерения;

– кнопку с надписью «Создать», при нажатии на которую будет создан и записан в массив очередной объект указанного в задании класса; предварительно выполнить проверку корректности заполнения пользователем полей для ввода значений;

– кнопку «Площадь», в обработчике события нажатия на которую вычислить и отобразить на форме результат суммы площадей всех созданных объектов (с указанием единиц измерения).

При добавлении объекта на форме должны быть отображены значения его характеристик (значения полей и значения, вычисленные в методах, площади и др.) с указанием единиц измерения без удаления информации о созданных ранее объектах.

Объявить массив из пяти объектов указанного в задании класса-наследника.

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

Варианты заданий

1. Описать классы Квадрат и Прямоугольник – наследники класса Квадрат.

Класс «Прямоугольник» должен содержать:

– поля и свойства для хранения информации о его сторонах;

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

– конструктор с параметрами (два параметра – стороны прямоугольника).

2. Описать классы Квадрат и Прямоугольный параллелепипед – наследники класса Квадрат.

Класс Прямоугольный параллелепипед должен содержать:

– 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);

– конструктор с параметрами (два параметра – стороны прямоугольника).

3. 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 соответствующие значения, а не отображать их на форме);

– конструктор с параметрами.

4. Описать базовый класс Круг и класс-наследник Сфера.

Класс-наследник должен содержать:

– поле и свойство для хранения информации о радиусе сферы;

– методы вычисления площади поверхности сферы, объема сферы (методы должны возвращать значения в точку вызова, а не отображать их на форме);

– конструктор с параметрами.

ЗАКЛЮЧЕНИЕ

При подготовке современных специалистов в области прикладной математики, информационных технологий, лазерной техники невозможно обойти стороной дисциплины, связанные с программированием и получением навыков в области разработки, отладки, тестирования и модификации программного обеспечения.

Приобретение навыков объектно ориентированного программирования даёт будущим специалистам возможность самостоятельной разработки классов, создания визуальных приложений, использования и доработки ранее написанного кода, в том числе и другими разработчиками, для решения широкого спектра учебных и профессиональных задач.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

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

2. Павловская, Т. А. П12 C++. Объектно-ориентированное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2006. – 265 с. – 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.

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

6. Фленов, М. Библия C#. – 4-е изд., перераб. и доп. – 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 ПМИ-125
Иванов И.И.
Принял:
ст. преподаватель
каф. ФиПМ
Шишкина М. В.

Владимир 2025

Приложение 2

Требования к форматированию отчёта по лабораторной работе

Отчёт по лабораторной работе должен быть создан в текстовом процессоре Microsoft Word. Печать отчета должна быть выполнена 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 прил. 1.

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

Поля: левое – 30 мм; правое – 10 мм; верхнее и нижнее – 20 мм.

Междустрочный интервал – полуторный.

Режим выравнивания – «по ширине».

Отступ в начале абзаца (красная строка) – 15 мм.

Все листы должны быть пронумерованы. Нумерацию начинать с титульного листа, но не проставлять на нём номер. Таким образом, номер страницы, следующей после титульного листа – 2. Номер проставить по центру страницы, кегль 12 пт.

Для допуска к защите лабораторной работы необходимо продемонстрировать работоспособность написанной программы, представить отчёт, написанный и оформленный в соответствии с требованиями, изложенными в прил. 3.

Для защиты работы необходимо ответить на вопросы по теоретической части работы, уметь 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 прил. 1.

2. Цель работы.

3. Постановку задачи. Формулировка цели работы и постановка задачи должны точно соответствовать указанным преподавателем.

4. Пять-семь ключевых слов на русском и английском языках. В качестве ключевых стоит выбирать слова, опираясь на которые можно составить исчерпывающий ответ по теме работы.

5. Краткую теоретическую часть, которая должна содержать от двух до четырёх страниц связного самостоятельно написанного текста с примерами. Содержание теоретической части должно быть основано на лекционном материале, но не должно быть его копией. Необходимо рассмотреть все вопросы, затронутые в работе, с теоретической точки зрения и на примерах, реализованных автором.

6. Ход работы должен содержать написанный программный код на изучаемом языке программирования и несколько тестовых примеров с пояснением полученных результатов.

7. Выводы по работе. В этой части необходимо указать, к какому выводу пришёл автор во время выполнения работы, рассказать о достоинствах и недостатках изученного метода, разработанного алгоритма, для решения какого класса задач используют этот подход.

Приложение 4

Типы данных языка C#

В основе любого типа в системе типов *.NET* (в том числе фундаментальных типов данных) лежит базовый класс `System.Object`.

Тип значений *(хранят значение)*	Ссылочный тип *(хранят ссылку на значение)*
– целочисленные типы	– строки
– вещественные	– массивы
– символы	– объекты
– 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

В языке *C#* отсутствует автоматическое преобразование символьных значений в целочисленные и обратно.

В языке программирования 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#

Так как C# строго типизированный язык, выполнение операций возможно только на однотипных переменных.

Для корректного использования в одном выражении разнотипных переменных прибегают к приведению типов.

Неявное приведение типов происходит автоматически при совместимости типов и если диапазон представления значений типа переменной-приёмника шире, чем у типа переменной-источника.

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

При необходимости присвоения переменной `s` значения переменной `i` нужно 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` для встроенных числовых типов, например `Int32.Parse`.

При преобразовании строки в число используют методы `Parse` или `TryParse`. Метод `Parse` следует использовать, если у программиста есть уверенность, что такое преобразование возможно

```
byte b = byte.Parse("7");
```

При обработке данных, введённых пользователем, рекомендуется использовать метод `TryParse`

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

здесь `s` – это строка, содержащая преобразуемое число, `result` – переменная числового типа, конкретный тип этой переменной определяется классом, метод которого вызван

```
if (double.TryParse(value, out number))
Console.WriteLine("'{0}' --> {1}", value, number);
```

Пример использования методов класса `Convert`:

```
int var1 = 8, var2 = 5;
byte sum = Convert.ToByte(var1+var2);
```

Приложение 6

Работа с массивами в языке C#

Под массивом понимают 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>;
```

Пример объявления массива: `int [] mas;`

При создании экземпляра массива используют оператор ***new***

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

Пример создания `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 следующими способами:

– вручную, обращаясь к конкретному элементу;

– при циклическом переборе элементов;

– при объявлении массива.

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

Каждый массив имеет свойство `Length`, хранящее максимально возможное количество элементов этого массива. Это свойство только для чтения. Свойство `Length` часто используют для организации циклического перебора элементов массива

```
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); }
}
```

В примере создан объект класса `Random` и использован его метод `Next`, возвращающий значение из указанного диапазона.

Оператор `foreach` позволяет организовать перебор элементов коллекции

```
foreach (<тип имя переменной цикла> in <коллекция>)
<оператор;>
foreach (int item in mas) Console.WriteLine("elem ={0}", item);
```

Учебное электронное издание

ШИШКИНА Мария Викторовна

ОБЪЕКТНО ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ

Практикум

Редактор А. П. Володина
Технический редактор Ш. Ш. Амирсейидов
Компьютерная верстка П. А. Некрасова
Корректор Н. В. Пустовойтова
Выпускающий редактор А. А. Амирсейидова

Системные требования: Intel от 1,3 ГГц; Windows XP/7/8/10; Adobe Reader; дисковод CD-ROM.

Тираж 9 экз.

Издательство Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.