**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**КЫРГЫЗСКОЙ РЕСПУБЛИКИ**

**КЫРГЫЗСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ им. И. РАЗЗАКОВА**

**Кафедра «Программное обеспечение компьютерных систем»**

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С++**

**Методические указания к выполнению**

**лабораторных работ для студентов направления 710400**

**«Программная инженерия»**

**Бишкек 2021**

РАССМОТРЕНО ОДОБРЕНО

на заседании кафедры Методическим советом ФИТ

«Программное обеспечение

Компьютерных систем»

Прот. № 4 от 28.10.2021 Прот. № от

УДК 004.43

Составил Каткова С.Н.

Объектно-ориентированное программирование: Методические указания к выполнению лабораторных работ для студентов направления 710400 «Программная инженерия»/ КГТУ им. И. Раззакова; сост. Каткова С.Н., Бишкек: ИЦ «Текник», 2016.- 48 с.

Методические указания содержат задания на разработку программ, рекомендации к выполнению, краткие теоретические сведения, примеры программ на языке C++ с применением технологии объектно-ориентированного программирования.

Предназначено для студентов направления «Программная инженерия» всех форм обучения.

СОДЕРЖАНИЕ

Введение ...................................................................................................................................4

Порядок выполнения работы...................................................................................................4

Требования к содержанию и оформлению отчета.................. …………………..................4

Лаб. раб. №1 Повторение.........................................................................................................5

Лаб. раб. №2 Классы и объекты...............................................................................................5

Лаб. раб. №3 Конструкторы и деструкторы............................................................................8

Лаб. раб. №4 Дружественные функции и классы................................................................13

Лаб. раб. №5 Перегрузка операторов....................................................................................17

Лаб. раб. №6 Наследование....................................................................................................21

Лаб. раб. №7 Виртуальные методы………….......................................................................30

Лаб. раб. №8 Шаблоны...........................................................................................................34

Лаб. раб. №9 Исключения.......................................................................................................39

Лаб. раб. №10 Статические поля и методы класса.................................................................45

Список литературы………. .....................................................................................................48

**Введение**

Данные методические указания предназначены для выполнения лабораторного практикума студентов, обучающихся по направлению «Программная инженерия» по дисциплине «Объектно-ориентированное программирование» и реализации на языке С++ основных концепций объектно-ориентированного программирования: инкапсуляции, наследования и полиморфизма. Указания содержат задания на разработку программ, рекомендации к выполнению, краткие теоретические сведения, примеры программ на языке C++ с применением технологии объектно-ориентированного программирования. Предложенные задания можно выполнять, используя различные компиляторы языка C++, поддерживающие ANSI-стандарт языка. Для разработки программ рекомендуется использовать различные интегрированные среды (студии) разработчика, например, Microsoft Visual Studio 2017 и выше для Windows. Лабораторный практикум имеет целью выработать у студентов навыки:

1. в применении языка программирования С++ как инструмента для изучения курса;
2. в изучении и использовании основных понятий и возможностей объектно-ориентированного стиля на языке программирования С++;
3. в разработке алгоритмов программ в виде блок-схем;
4. в разработке программного кода задач на основе блок-схем.

**Порядок выполнения работы**

1. Внимательно изучить методические указания к лабораторной работе.
2. Обсудить с преподавателем возникшие вопросы и показать, что как вы поняли постановку задачи.
3. Получив одобрение от преподавателя, приступить к выполнению задания.
4. Оформить отчет по лабораторной работе согласно требованиям к содержанию и оформлению.

**Требования к содержанию и оформлению отчета**

1. постановка задачи;
2. описание исходных данных программы: полей, методов, классов, объектов, возможно файлов;
3. разработка алгоритма задания в виде блок-схемы;
4. текст программы, выполненной на языке «С++» с обязательными комментариями;
5. результат работы программы в виде скриншотов, защита работы кодов преподавателю вживую.
6. список использованной литературы и источников.

К сдаче экзамена допускаются студенты, получившие удовлетворительные оценки по всем лабораторным работам.

**Лабораторная работа №1**

**Тема:** Повторить пройденный материал - типы данных, функции и структуры в программировании для лучшего понимания свойств классов в объектно-ориентированном программировании.

**Задания**

1. Возведение числа **n** в степень **p** – это умножение числа **n** на себя **p** раз.

Напишите функцию (не библиотечную) с именем **power** (), которая в качестве аргументов принимает значение типа **double** для **n** и значение типа **int** для **p** и возвращает значение типа **double**.

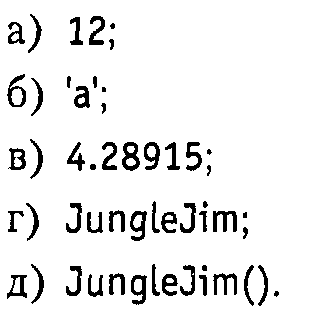
Для аргумента, соответствующего степени числа, задать значение по умолчанию, равное **2**, чтобы при отсутствии показателя степени при вызове функции число **n** возводилось в квадрат.

Сделайте проверку, будет ли пользователь вводить степень числа. Если будет, то эта степень – число, большее 2. Напишите функцию **main**(), которая запрашивает у пользователя ввод аргументов для функции **power** () и отображает на экране результаты её работы. Напишите код первого задания с *объявлением* (прототипом) функции power ().

1. Взяв в качестве основы функцию **power** () из первого задания, работающую только со значением типа **double**. Создайте перегруженные функции с этим же именем, принимающими в качестве аргумента значения типа **char**, short **int**, **long int, float**. Напишите программу, вызывающую функцию **power** () со всеми возможными типами аргументов. Напишите код второго задания без *объявления* функции power ().
2. Информация по работникам N заводов задается структурой с полями: фамилия, возраст, специальность, средний оклад. Ввести информацию по заводам, посчитать количество слесарей и токарей. Вывести эти значения на консоль.

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №1**

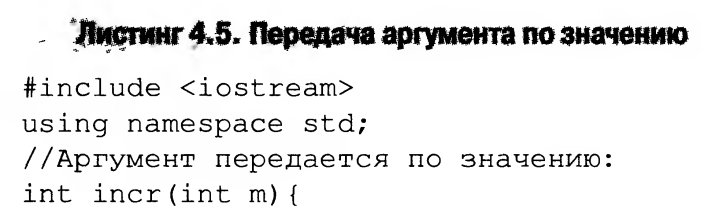
1. Какие стандартные типы используются в С++? Сколько под них резервируется памяти?
2. К каким элементам программы относятся следующие:

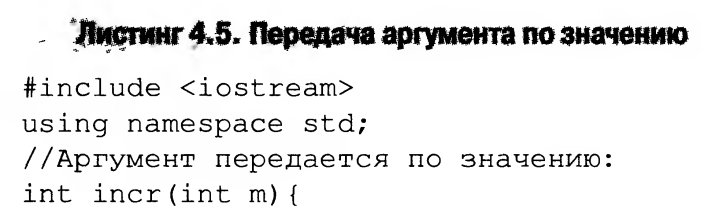


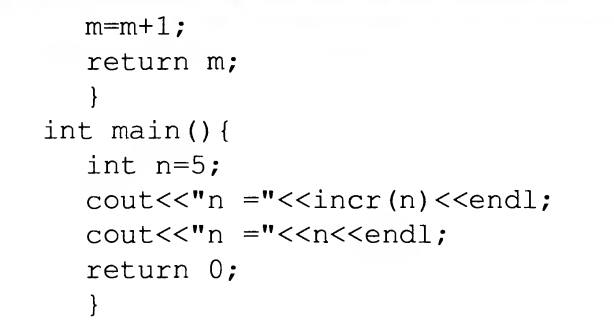
1. Что такое функция? Какова роль функций в языке С++?
2. Напишите синтаксис функции.
3. Опишите способы использования функций в программах с объявлением функций и без объявления функций.
4. Объясните механизмы передачи аргументов по значению и по ссылке в функцию.

Объясните результаты работы программ.

1)







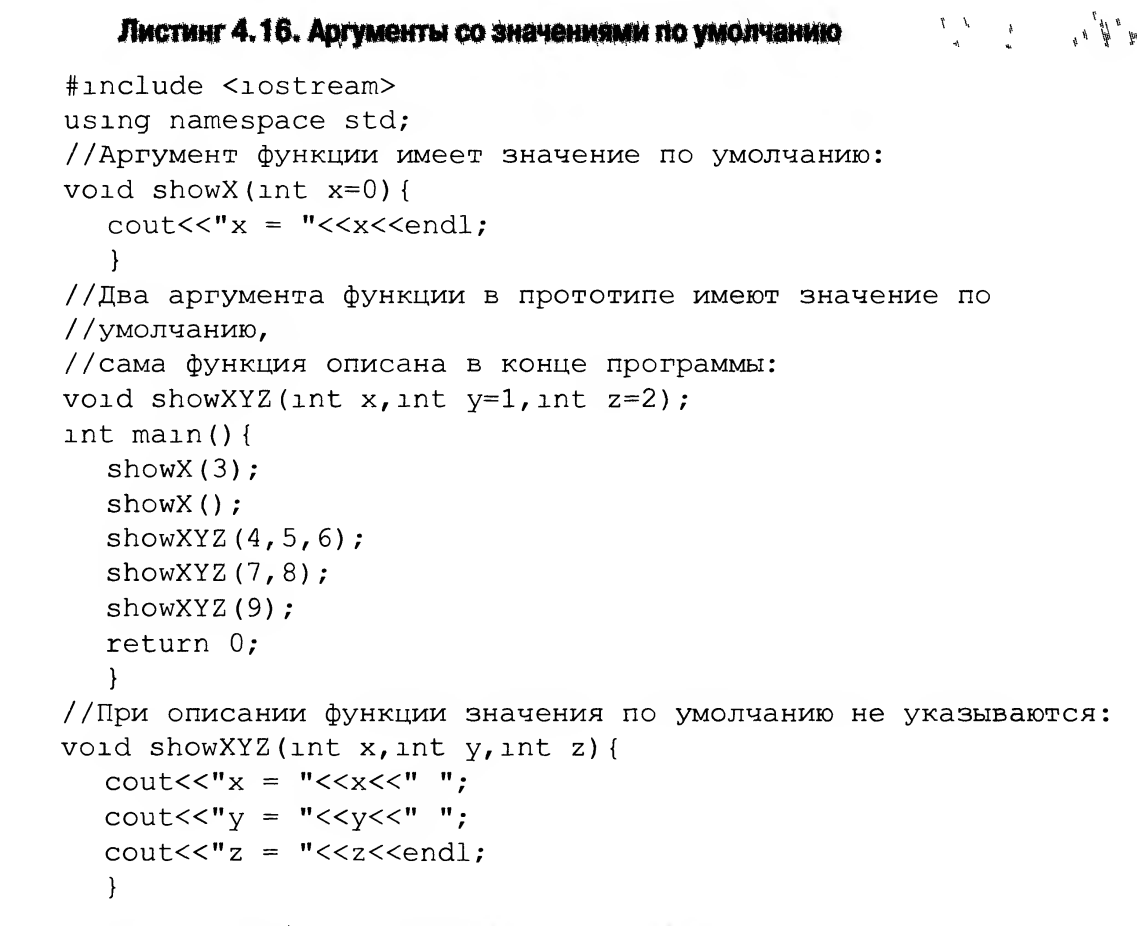
2))





1. Назовите разновидности аргументов, которые могут быть переданы параметрам функции?
2. Что такое аргументы по умолчанию?

Объясните результат работы программы:

,

1. Что такое перегрузка функций?
2. Чем отличается структура (struct) от массива?

**Лабораторная работа №2**

**Тема: Классы и объекты**

**Задания**

1. В здании аэропорта на экранах отображается информация о самолетах, а именно: информация о пункте отправления, пункте назначения, номере рейса, времени прибытия, времени отправления, номере секции для регистрации. Экраны – это средство, которое помогает своевременно зарегистрировать и отправить пассажиров. Важно, чтобы информация на экранах была понятной и верной.

Создайте необходимую информацию в виде таблицы для такого экрана, с помощью класса Aeroflot, содержащего в описании следующие поля: номер рейса; название пункта отправления; название пункта назначения рейса; время прибытия; время отправления; место регистрации.

Напишите код программы, выполняющей следующие действия: ввод с клавиатуры значений полей объектов; сортировку записей в таблице в алфавитном порядке по названию пунктов назначения; вывод на консоль значений полей объектов класса в виде таблицы рейсов; если таких рейсов нет, выдать соответствующее сообщение.

**2. Опишите класс «студенческая группа».**

Предусмотрите возможность: работы с переменным числом студентов; поиска студента по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона); добавления или удаления записей; сортировки по разным полям.

Разработайте программу, демонстрирующую работу с этим классом.

Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

**3. Опишите класс, реализующий стек (Stack).**

Разработать программу, использующую этот класс для моделирования Т-образного сортировочного узла на железной дороге. Программа должна разделять на два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность формирования состава из файла или с клавиатуры. Возможно использование стандартных функций при работе со стеком в виде контейнера из библиотеки STL:

**push()** - поместить элемент в вершину стека;

**pop()** - удалить элемент из вершины стека;

**Дополнительное задание**

1. Разработайте спецификации требований к ПО, которые решают эти задачи.
2. Спроектируйте ПО: входной и выходной интерфейсы для консоли, блок-схему программ.
3. Определитесь со структурой программы (прототипы, описание, вызов функций), разработайте код на языке объектно-ориентированного программирования С++, протестируйте и отладьте его.

**Методические указания**

Для выполнения лабораторной работы №1 требуется знать основные понятия *класс*, *элементы* класса, *инкапсуляция****,****правило определения классов*, *разграничение* *доступа* *к элементам класса*, *объект*, *правило* *создания объектов* *класса*.

**Класс (class)** - это абстрактный тип данных, заданный программистом. Класс описывает структуру своих объектов (поля) и их поведение (методы). Класс представляет модель реального объекта в виде данных и функций для работы с ними. Данные класса называются полями, а функции класса методами.

Поля и методы класса называют **членами** или **элементами класса**.

Класс позволяет объединить данные и оперирующие ими функции в одной структуре.

Такое объединение обычно называют инкапсуляцией.

**Инкапсуляция** – это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то и другое от внешнего вмешательства или неправильного использования. Позволяет скрыть конкретную реализацию класса, облегчая отладку и модификацию программ.

**Смысл определения класса**: фактически определение класса задает внутреннюю структуру объектов класса, т.е. из каких полей они состоят и какими методами пользуется объект класса при работе с этими полями. При определении класса не происходит выделения физической памяти. Память отводится при объявлении (создании) объекта.Общий вид синтаксиса класса**:**

*// имя должно быть уникальным среди имен других классов, определенных в вашей программе.*

**сlass <**имя\_ класса**>**

**{**

**private:**

<сокрытые поля и методы класса>

**public:**

<общедоступные поля и методыкласса**>**

**} [**список объектов]**;**

Список объектов можно определить позднее в функции main().

**Разграничение доступ к полям и методам** класса можно регулировать с помощью идентификаторов доступа **private** и **public**. Элементы, описанные после служебного слова **private,** доступнытолько внутри класса. Этот вид доступа принят в классе по умолчанию (т.е. все элементы класса – данные и методы будут скрыты по умолчанию). К закрытым элементам класса можно получить доступ только с помощью методов самого класса. Элементы, описанные после служебного слова **public,** доступныи за пределами класса. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций **private** и **public,** порядок их следования значения не имеет. Пример описания классаСomponents. **(***Известно, что завод выпускает разные модели деталей для комплектующих изделий***).**

class Сomponents

{

private:

float Cost; //сокрытое поле-элемент класса

public:

int СomponentNumber; //общедоступное поле-элемент класса

int ModelNumber; //общедоступное поле-элемент класса

void SetCost (float c) // общедоступный метод-элемент класса

{Cost=c;}

void ShowCost () // общедоступный метод-элемент класса

{cout<<"Cost: "<<Cost<<endl;}

};

Методы void SetCost (float c) и void ShowCost () являются встроенными, так как их тела определены внутри класса. Встроенные методы делают короткими. Если внутри класса записан только заголовок метода, сам метод должен быть определен в другом месте программы. Когда определяется метод отдельно от класса, то имени метода должно предшествовать имя класса и оператор разрешения области видимости (::)

**Объект (object)** - это экземпляр класса, т.е. его физическая реализация. Практически класс может пониматься как некий шаблон, по которому создаются объекты — экземпляры данного класса. Все экземпляры одного класса созданы по одному шаблону, поэтому имеют один и тот же набор полей и методов. Значения этих полей разные для разных объектов.

Методы класса разные объекты используют одни и те же. В программировании объект– это переменная класса, а класс – это новый пользовательский тип переменной.

**Объявление объектов**

Два объекта part1, part2 класса Сomponents объявлены в первом операторе функции main (). При определении класса Сomponentsмы не создали объекты. Определение класса лишь задает структуру будущего объекта. Все операции программа производит с объектами. Объявление объекта похоже на определение переменной, оно означает выделение памяти, необходимой для хранения объекта:

Сomponents part1, part2; // объявление двух объектов

В результате отводится память под одноименные поля двух объектов.

2

СompnentsNumber

ModelNumber

5

250.0

Сost

ModelNumber

4

СomponentsNumber

15

Сost

180.0

**part1**

**part2**

Объекты класса **Сomponents**

Рис. 1 Два объекта класса Сomponents

При определении объектов под три поля каждого объекта выделяется одинаковая память. Имена полей одинаковые, а значения – разные. Под методы память отводится один раз. Оба объекта пользуются одними методами.

part1

part2

Cost

Cost

ModelNumber

Cost

ModelNumber

Cost

СompentsNumber

СompentsNumber

SetCost()

ShowCost()

Рис.2 Объекты, данные, функции и память

Доступ к открытым полям объекта класса **напрямую** через операцию (точка):

part1.ModelNumber=2;

part1. ComponentNumber =5;

Чтобы обратиться к полю, надо указать имя объекта класса, которому принадлежит поле, а затем через точку – имя поля.

Доступ к закрытым полям объекта класса **только через методы** объекта:

part1.ShowCost();

Чтобы вызвать метод, надо указать имя объекта класса, для которого будет вызван метод, а затем через точку – имя метода.

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №2**

1. Как вы понимаете, что такое класс и объект? Приведите примеры.
2. Каков синтаксис и смысл определения класса? Как называются составные части класса?
3. Что такое идентификаторы доступа к элементам класса? Дайте определения.
4. Что такое инкапсуляция?
5. Где в программе с классами можно создавать объекты?
6. Каков смысл объявления объектов?
7. В какой момент метод готов для вызова объектом?
8. Как называются методы, встроенные в структуру класса?
9. Как определяется метод, если внутри класса записан только его заголовок, сам метод определен вне класса?
10. Если в классе два поля данных и два объекта, сколько полей принадлежит каждому объекту? Совпадет ли имена и значения этих полей для объектов?
11. Тиражируются ли методы класса?
12. Как в программе написать доступ к открытым и закрытым полям класса?

**Лабораторная работа № 3**

**Тема: Конструктор и деструктор**

**Задания:**

1. Разработайте программу с классом **Timer**, в которой таймер действует как таймер обратного отсчета. При создании объекта типа **timer** ему присваивается начальное значение времени. В результате вызова функции **run()** таймер начинает отсчет в сторону уменьшающихся значений, пока не достигнет значения 0, после чего зазвонит звонок. **Создайте 3 варианта конструктора, т.е. конструктор перегрузите 3 раза** для того, чтобы можно было указывать время в секундах с помощью целого числа или строки, или в минутах и секундах, если указываются два целых числа.

Эта программа использует библиотечную функцию **clock(),** возвращающую число тиков, прошедших с момента запуска программы. Поделив это значение на макрос **CLK\_TCK**, получаем значение в секундах. Прототипы для **clock**() и **CLK\_TCK** содержатся в заголовочном файле time.h.

2. Создайте класс **Nomenclature**, описывающий товары на складе магазина. Закрытыми элементами класса будут: название товара, оптовая цена, розничная наценка и количество товаров на складе. Включите в класс открытые функции подсчета возможного чистого дохода при продаже этого товара и вывода всех данных о товаре на экран. Для инициализации и удаления объектов класса используйте конструкторы и деструкторы.

3. Создайте класс **Soft**, который содержит информацию об установленном программном обеспечении. Закрытыми элементами класса будут: название программы, разработчик, занимаемый объем, дата завершения лицензии. Включите в класс открытые функции подсчета количества дней до завершения лицензии и вывода всех данных об установленном программном обеспечении на экран. Для инициализации и удаления объектов класса используйте конструкторы и деструкторы.

**Дополнительное задание**

1. Разработайте спецификации требований к ПО, которые решают эти задачи.
2. Спроектируйте ПО: входной и выходной интерфейсы для консоли, блок-схему программ.
3. Определитесь со структурой программы (прототипы, описание, вызов функций), разработайте код на языке объектно-ориентированного программирования С++, протестируйте и отладьте его.

**Методические указания**

Для выполнения лабораторной работы №2 требуется знать основные понятия *конструктор*, *деструктор, конструктор по умолчанию, конструктор с аргументами, перегрузка конструктора*, *конструктор копирования*.

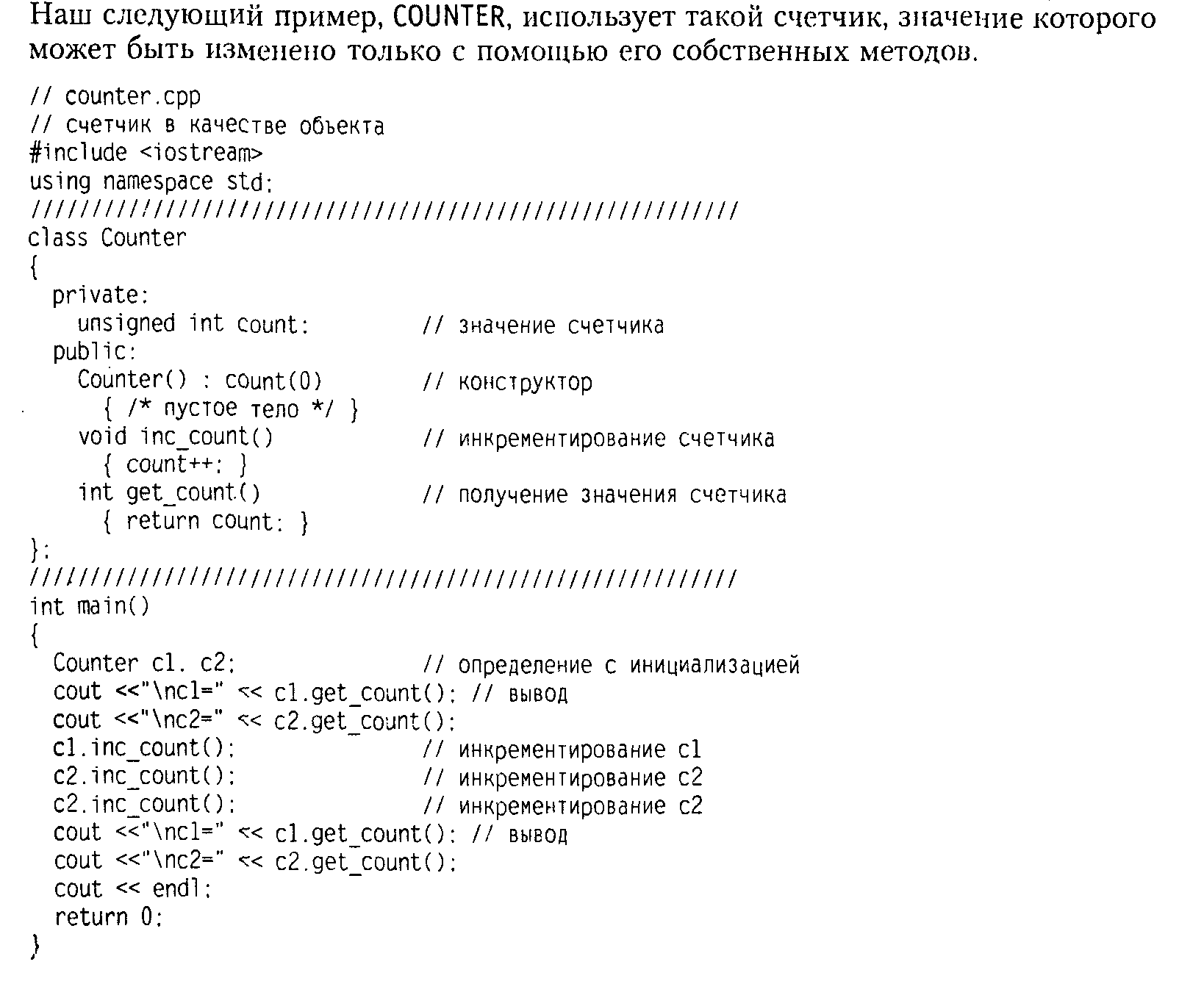
**Конструктор** - это метод класса, выполняющийся автоматически в момент создания объекта. Конструктор предназначен для инициализации объекта.

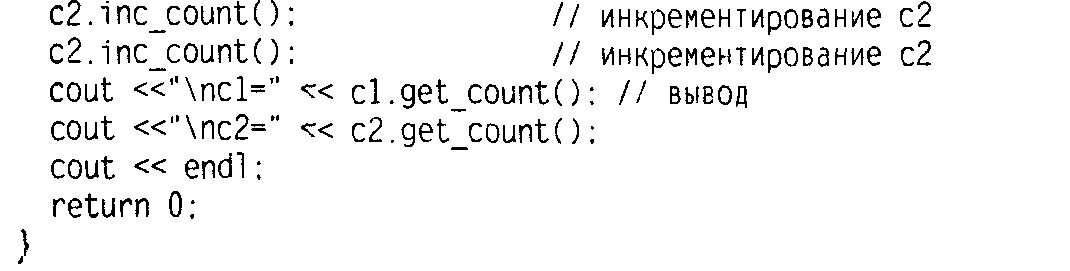
**Особенности конструктора:**

1. Имя конструктора совпадает с именем класса.
2. У конструкторов не существует возвращаемых значений.
3. Отсутствие типа возвращаемого значения. Нельзя получить указатель на конструктор.
4. Если пользователь не указал ни одного конструктора, компилятор создает его автоматически.
5. Конструктор не наследуется.
6. Конструктор можно перегружать по разным аргументам.
7. Существует конструктор по умолчанию (без аргументов) и конструктор с аргументами.

**Конструктор по умолчанию** - это конструктор, который может быть вызван без передачи аргументов.

Например, в качестве объекта возьмем счетчик **count** класса **Counter**, значения которого могут быть изменены только с помощью собственных методов.

****

****

Класс **Counter** имеет единственное поле count типа unsigned **int**, поскольку значение счетчика не может быть отрицательным, и три метода: конструктор **Counter** (), **inc\_count(),** инкрементирующий поле **count, и get\_count(),** возвращающий текущее значение счетчика.

Для первоначальной инициализации счетчика нулевым значением можно использовать конструктор по умолчанию:

***Counter(): count(0)***

***{/пустое тело/};***

Как работает этот конструктор?

Пусть в функции main () создаются два объекта с1 и с2 с помощью оператора определения:

***Counter с1, с2;***

Данный оператор кроме создания объектов, присваивает полю каждого из объектов нулевое значение. Так как при создании **каждого объекта**!!! автоматически вызывается конструктор Counter () и он выполняет автоматическую инициализацию поля каждого объекта нулевым значением в соответствии с программным кодом. Сам по себе конструктор никаких значений полям не присваивает, он просто приходит в активное состояние с помощью компилятора при создании объекта и ждет команд от программиста по инициализации полей. Конструктор инициализирует поля в соответствии с программным кодом. Без помощи кода он активен, но инициализацию не выполняет.

**Результаты работы программы**

В функции main () рассматриваемой программы создаются два объекта класса Counter с именами с1 и с2. Затем на экран выводятся значения полей каждого из объектов, которые, согласно нашей задумке, должны быть инициализированы нулевыми значениями. Далее значение счетчика с1 инкрементируется один раз. А значение счетчика с2 – два раза, и программа вновь заставляет объекты вывести значения своих полей на экран (что является в данном случае вполне корректным). Результат работы программы:

с1=0

с2=0

с1=1

с2=2

Из результатов видно, что конструктор **Counter** () действительно выполняет автоматическую инициализацию поля каждого объекта нулевым значением в соответствии со значением, записанным в круглые скобки в определении инициализируемого поля конструктора **count (0).**

**Способы инициализации в конструкторе по умолчанию**

1. Инициализация полей объекта с помощью списка инициализации:

***Counter(): count(0)***

***{ };***

Список инициализации находится после двоеточия. Инициализирующее значение находится в скобках после имени поля.

1. Инициализация полей объекта с помощью присвоения им значений в теле конструктора. Её в нашем примере можно выполнить так:

***Counter ()***

***{count =0;}***

Второй способ не содержит ошибок, но он не рекомендуется.

Причины:

1. Инициализация полей должна происходить до начала исполнения тела конструктора. Чтобы в теле конструктора можно было производить более сложные действия, чем обычная инициализация.
2. Список инициализации - это единственный способ задать первоначальные значения константам и ссылкам.

**Конструктор с аргументами** – это конструктор, имеющий хотя бы один аргумент, может инициализировать поля значениями, переданными ему в качестве аргументов. К тому же этот конструктор значительно упрощает код программы.

*Код инициализации полей класса с использованием метода Set():*

void SetComponents (int mn, int cn, float c) // метод

{

ModelNumber=mn; // инициализирует

ComponentsNumber=cn; //3 поля

Cost=c; // объекта

}

*В функции main пишем три команды при использовании метода Set():*

Сomponents part1, part2;

part1.SetComponents(2,5,250.0);

part2.SetComponents (4, 15, 180.0);

**Сравните!!!**

*Код инициализации полей класса с использованием метода “конструктор с аргументами”:*

Сomponents (float c, int mn, int cn,): // конструктор инициализирует

Cost(c), ModelNumber(mn), ComponentsNumber(cn)// 3 поля объекта

{ }

*В функции main () пишем одну команду**при использовании метода “конструктор с аргументами” для инициализации трех полей объекта. :*

Сomponents part1(2,5,250.0), part2(4, 15, 180.0);

На основе рассмотренного примера делаем вывод о том, что код программы значительно сокращается при использовании конструктора с аргументами.

**Деструктор** – это метод, автоматически вызываемый при уничтожении объекта.

Особенности деструктора:

1. Имеет имя, совпадающее с именем конструктора, а, следовательно, и класса.
2. Предваряется символом тильда.
3. Не возвращает значения.
4. Не имеет аргументов

Сomponents(float c,int mn, int cn): // конструктор с аргументами

Cost(c),ModelNumber(mn),ComponentsNumber(cn)

{ }

~ Components ()

{ }

Конструкторы базовых классов вызываются в порядке их объявления. Деструкторы вызываются в обратном порядке.

**Перегрузка конструктора**

Рассмотрим два основных способа перегрузкиконструктора:

**способ 1**. Использование в одной программе обоих типов конструктора.

class Сomponents{

……..

Сomponents():Cost(0), ModelNumber(0), ComponentsNumber(0){} //конструктор по умолчанию

Сomponents(float c,int mn, int cn): // конструктор c аргументами

Cost(c), ModelNumber(mn),ComponentsNumber(cn){}

}; ……..

int main()

{

Сomponents part1, part2(2,5,250.0); // объявление двух объектов

part1.ShowComponents(); //вызов метода выдачи на экран для первого объекта

part2.ShowComponents (); //вызов метода выдачи на экран для первого объекта

}

Для первого объекта **part1** значения всех полей объекта нулевые, так как инициализация его полей осуществляется с помощью конструктора по умолчанию (без аргументов.) нулями.

Для второго объекта **part2** значения полей будут соответственно равны: Cost=2,5; ModelNumber=5; ComponentsNumber=250.0. так как инициализация его полей осуществляется с помощью конструктора этими значениями аргументов.

**Способ 2**. Для перегрузки используются конструкторы с одним именем, но отличающиеся количеством аргументов.

Рассмотрим класс MyClass, с закрытыми полями m, n. Для инициализации этих полей используем три конструктора: без аргументов, с одним аргументом, с двумя аргументами.

//Конструктор без аргументов инициализирует поля нулевым значением

MyClass(): m(0),n(0){}

//Конструктор с одним аргументом инициализирует поля одним значением

MyClass(int a): m(a),n(a){}

//Конструктор с двумя аргументами инициализирует поля разными значениями

MyClass(int a, int b): m(a),n(b){}

~ MyClass(){}

**Конструктор копирования –** это конструктор, с помощью которого возможно сделать третий способ инициализации полей объекта. При этом способе для инициализации полей объекта используются поля уже существующего объекта. Для этого не нужно самим создавать специальный конструктор, поскольку такой конструктор предоставляется компилятором для каждого создаваемого класса и называется конструктором копирования по умолчанию. Копирующий конструктор имеет единственный аргумент, являющийся объектом того же класса, что и конструктор.

class Сomponents{

……..

Сomponents():Cost(0), ModelNumber(0), ComponentsNumber(0){} //конструктор по умолчанию

Сomponents(float c,int mn, int cn): // конструктор c аргументами

Cost(c), ModelNumber(mn),ComponentsNumber(cn){}

}; ……..

int main()

{

Сomponents part1, part2(2,5,250.0); // определение двух объектов

part1.ShowComponents(); //вызов метода выдачи на экран для первого объекта

part2.ShowComponents (); //вызов метода выдачи на экран для первого объекта

}

Добавим в этот программный команду конструктора копирования:

Сomponents part3(part2);

В результате создан объект part3, имеющий те же поля, что объект part2.

Возможен и такой вариант команды конструктора копирования:

Сomponents part3= part2;

Результат работы этих команд одинаков.

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №3**

1. Что такое конструктор? Как он вызывается на выполнение?
2. Особенности конструктора, отличающие его от других методов класса?
3. Что такое конструктор по умолчанию? В каком случае следует его использовать?
4. Способы инициализации в конструкторе по умолчанию? Какой способ инициализации предпочтительнее?
5. Что такое конструктор с аргументами? Когда используют конструктор с аргументами?
6. Какие преимущества у конструктора с аргументами перед методом Set()?
7. Что такое деструктор? Для чего его используют?
8. Особенности деструктор
9. В какой последовательности выполняется конструкторы, в какой деструкторы классов?
10. Можно ли перегружать конструктор?
11. Что такое конструктор копирования?

**Лабораторная работа №4**

**Тема: Дружественные функции и дружественные классы**

**Задания:**

1. Для участия в ежегодном трансконтинентальном Ралли-марафоне «Дакар» по усложненной трассе в Южной Америке выбраны грузовики [КАМАЗ](https://ru.wikipedia.org/wiki/%D0%9A%D0%90%D0%9C%D0%90%D0%97) и [Tatra](https://ru.wikipedia.org/wiki/Tatra), которые после 2000 года лидируют в зачете грузовиков.

Решите задачу сравнения скоростей движения грузовиков по разным по проходимости участкам трассы, а именно: по равнине, горам, пустыне. Создайте и выдайте на экран таблицу результатов ралли - марафона. Определите победителя.

Для решения задачи используйте классы **Kamaz** и **Tatra,** а такжефункциюсравнения скоростей **FrCreater (**). Функция  **FrCreater (**) возвращает число +1, если объект **kamaz** движется быстрее объекта **tatra**; нуль, если их скорости одинаковы; число -1, если объект **kamaz** движется медленнее объекта **tatra** .

Оба класса содержат поля: «скорость» и «наименование» грузовой машины, а также методы: инициализация и отображение полей на экране. Определитесь с идентификаторами доступа к членам класса, не нарушая принципа инкапсуляции.

1. Сделайте функцию **FrCreater(**) дружественной: классу **Kamaz,** обоим классам **Kamaz** и **Tatra.**
2. Сделайте класс **Tatra** дружественным классу **Kamaz.**

**Дополнительное задание**

1. Разработайте спецификации требований к ПО, которые решают эти задачи.
2. Спроектируйте ПО: входной и выходной интерфейсы для консоли, блок-схему программ.
3. Определитесь со структурой программы (прототипы, описание, вызов функций), разработайте код на языке объектно-ориентированного программирования С++, протестируйте и отладьте его.

**Методические указания**

Для выполнения лабораторной работы №3 требуется знать основные понятия *дружественная функция; функция, дружественная одному классу; функция, дружественная двум классам и более; дружественные классы.*

В каком случае нужны *Дружественные функции?* Когда необходимо получить прямой доступ к внутренним данным одного или более классов. *Дружественные функции – это мосты между классами (Р.* Лафоре).

**Дружественная функция**—это функция, не являющаяся членом класса, но обеспечивающая доступ ко всем членам класса, для которого она дружественна. Независима от класса **дружественная функция** находится вне класса. Но она объявляется внутри класса как обычная функция и перед ней указывается ключевое слово **friend**. Вне класса перед описанием дружественной функции это ключевое слово не ставится**.**

**Синтаксис прототипа дружественной функции**

***friend*** тип имя ([список\_ параметров])

Рассмотрим пример функции, дружественной одному классу, которая имеет доступ к закрытым полям этого класса.

.

class MyClass{

//закрытое поле класса

double x;

public:

//Дружественная функция должна быть объявлена внутри класса (прототип д.ф.)

friend void show (MyClass obj);

};

// Описание дружественной функции следует делать вне класса

void show (MyClass obj){

cout <<”x = ”<< obj.x<<endl;)

int main(){

MyClass.a(10);

//Дружественная функция имеет доступ к закрытым члена класса

show(a);

return 0;

}

В данном случае поле х класса MyClass является закрытым, поэтому получить доступ к этому полю класса можно только внутри класса. для инициализации этого поля используем конструктор, которому передается один аргумент. Других методов доступа в классе нет. С помощью внешней функции show() отобразим значение поля класса на экране. Чтобы эта внешняя функция имела доступ к закрытому полю, она объявлена, как дружественная. Для этого в описании класса добавлен её прототип friend void show (MyClass obj). Благодаря тому, что дружественная функция дружественна к классу, корректной является команда **show(а)** с помощью которой отображается значение закрытого поля объекта **а**. Прототип (заголовок) дружественной функции указан в описании класса**.**

* Так как дружественная функция не является методом класса, для которого она дружественна, нельзя вызвать дружественную функцию обычным способом в точечном формате с помощью команды **a.show()**, а можно только командой **show(a).** Так как, хотя дружественная функция "знает" о закрытом поле класса, для которого она является дружественной, доступ к нему она может получить только через объект этого класса, который объявлен внутри функции и передан ей в качестве аргумента. ДФ вызывается для объекта класса.
* Дружественная функция не наследуется. То есть, если в базовый класс дружественная функция включается, то она не является таковой для производных классов.
* Дружественная функция может быть дружественной более чем к одному классу.

**Функция, дружественная двум классам и более**

Такая дружественная функция полезна тогда, когда у двух и более разных классов имеется нечто общее, с чем необходимо произвести какие-то действия, например, сравнение или арифметические действия закрытых полей объектов, принадлежащих разным классам.

Рассмотрим пример программы. Есть два класса А и В, у каждого из которых есть закрытое поле типа **double**. Функцией **summa** () вычисляется сумма закрытых полей объектов, переданных ей в качестве аргументов. Первый аргумент функции – объект класса А, второй аргумент функции – объект класса В. Чтобы функция имела доступ к полям обоих классов, в каждом классе эта функция объявлена, как дружественная. Кроме того, поскольку впервые прототип дружественной функции в программном коде появляется до описания класса В, до этого места в программе приведен анонс класса (команда class B перед объявлением класса А). Такой анонс необходим для того, чтобы идентифицировать как класс объявление типа во втором аргументе функции.

//Анонс класса:

class B;

//Класс с закрытым полем

class A{

double x;

public:

A (double z): x(z){}

// Прототип дружественной функции с двумя аргументами

friend double summa (A a, B b);

} a(3.5);

//Класс с закрытым полем

class B{

double y;

public:

B(double z): y(z){}

// Прототип дружественной функции с двумя аргументами

friend double summa (A a, B b); } a(2.3);

double summa(A a, B b){

return a.x+b.y;

}

int main(){

//Вызов дружественной функции

cout<<”Total is ”<<summa(a,b)<<endl;

return 0;

}

В результате выполнения программы получим:

Total is 5.8

**Дружественные классы**

Часто дружественные классы используются тогда, когда классу необходим доступ к закрытым данным другого класса. Рассмотрим случай, когда класс **Вeta** объявляется другом классу **Аlpha**.

class Alpha{

private:

int data1;

alpha():data1(99){} //конструктор по умолчанию

fiend class Beta //Дружественный класс Beta

};

//Описание дружественного класса Beta

class Beta{

public:

void func1 (Alpha a){cout “n\data1 = ”<< a. data1};

void func2 (Alpha a); ){cout “n\data1 = ”<< a. data1};

};

int main() {

Alpha a;

Beta b;

a.func1(a);

b.func2(a);

cout endl;

return 0; }

Для класса Alpha весь класс Beta провозглашен дружественным. Теперь все методы Beta имеют доступ к скрытым данным класса Alpha (в данной программе это лишь единственная переменная data1).

Для класса весь класс провозглашен дружественным. Теперь все методы beta имеют доступ к скрытым данным класса alpha (в этой программе это лишь единственная переменная data1). Мы объявили класс дружественным, используя выражение: Friend class Beta.

С одной стороны, дружественные функции повышают гибкость языка, но, с другой стороны, они не соответствуют принципу ограничения доступа к данным, в соответствии с которым только методы класса могут иметь доступ к сокрытым данным класса. Множество дружественных функций фактически стирают границы между классами, поэтому они должны быть разбросаны по программе и встречаться как можно реже.

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №4**

1. Что такое дружественная функция, для чего она нужна?
2. Как она объявляется, каков её синтаксис?
3. Через что дружественная функция может получить доступ к закрытому полю класса?
4. Какой вариант вызова ДФ правильный:
5. a.show() или show(a)?
6. Наследуется ли дружественная функция?
7. Может ли дружественная функция быть дружественной более чем к одному классу?
8. Для решения каких задачах есть смысл использовать функцию, дружественную двум и более классам?
9. Когда нужно использовать дружественные классы?
10. Какому принципу ООП не соответствуют дружественные функции?
11. Как можно ограничить использование ДФ?
12. В каких крайних случаях нужно использовать ДФ?

**Лабораторная работа № 5**

**Тема: Перегрузка операторов**

**Задания:**

1. Типы данных полезны там, где ошибки могут быть вызваны арифметическим переполнением, которое не допустимо.

Создайте и откомпилируйте класс **Int**. Перегрузите четыре бинарных целочисленных арифметических операции (+, -, \*, /) и унарные операции постфиксной и префиксной форм инкремента с помощью внутренней операторной функции так, чтобы их можно было использовать для операций с объектами класса **Int**.

Если результат какой-либо из них выходит за границы типа **int** (в 32-битной системе), имеющее значения от **2 147 483 648** до **-2 147 483 648**, то операция должна послать сообщение об ошибке и завершить программу. Для выявления ошибки арифметического переполнения используйте концепцию **исключения**.

Для облегчения проверки переполнения выполняйте вычисления с использованием типа **long** **double**. При описании унарных операций используйте указатель **this**.

1. Опишите класс **fraction**, у которого есть одно закрытое целочисленное поле **chislo** типа **double**. Перегрузите для этого класса арифметические операции **сложения**, **вычитания**, **умножения** и **деления** так, чтобы они могли оперировать как с объектами класса, так и с числами (то есть выполнять, например, не только действие 3/4 +2/5, но и 1/2 + 4 или 2\* 5/6). Также перегрузите унарную операцию инкремента в префиксной или постфиксной форме увеличения дроби. Продемонстрируйте работу класса. Используйте конструктор по умолчанию и конструктор с одни аргументом для инициализации поля класса.
2. Создать класс, в котором перегружается метод **rect\_area()**.

Этот метод возвращает площадь прямоугольника. В этой программе метод **rect\_area()** перегружается двумя способами. В первом — методу передаются оба размера фигуры. Эта версия используется для прямоугольника. Однако, в случае квадрата необходимо задавать только один аргумент, поэтому вызывается вторая версия метода **rect\_area().**

**Дополнительное задание**

1. Разработайте спецификации требований к ПО, которые решают эти задачи.
2. Спроектируйте ПО: входной и выходной интерфейсы для консоли, блок-схему программ.
3. Определитесь со структурой программы (прототипы, описание, вызов функций), разработайте код на языке объектно-ориентированного программирования С++, протестируйте и отладьте его.

**Методические указания**

Для выполнения лабораторной работы №4 требуется знать основные понятия *полиморфизм****,*** п*ерегрузка операторов,**операторная функция перегрузки, перегрузка бинарных и унарных операторов.*

В [языках программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) **полиморфизмом** называется способность функции обрабатывать данные разных [типов](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85). **Цель полиморфизма** - использование одного имени метода для задания общих для класса действий и тем самым снижение сложности программ. Пример полиморфизма в ООП – это перегрузка операторов.

**Перегрузка операторов** – это возможность заставить компилятор осуществлять стандартные операции над нестандартными (пользовательскими) типами данных.

Перегрузка операторовс помощью операторной функции operator выполняется по шагам:

1. Компилятор идет по коду (начиная с главного метода).
2. Находит арифметическое выражение.
3. Анализирует, знает ли он типы, над которыми совершается операция. Если типы ему известны (стандартные типа int, float и т.д.), то он выполняет действие.
4. Если типы ему неизвестны (пользовательские типы), он *ищет операторную функцию перегрузки* **operator** и затем успешно выполняет действие над значениями полей объектов с помощью операторной функции.

**Синтаксис** **объявления операторной функции перегрузки**

Тип\_результата **operator** символ\_ оператора (аргументы: тип и название)

{

программный код

}

Сначала указывается тип результата, который возвращается операторной функцией – как в случае объявления обычной функции. Ключевое слово *operator* является обязательным признаком операторной функции. После этого слова указывается непосредственно тот оператор, который переопределяется операторной функцией. Это может быть. Например, оператор +, -, \* или /, хотя этим набором допустимых для переопределения операторов не ограничивается. После оператора в скобках указывают аргументы операторной функции. Перед каждым аргументом указывается тип или класс, к которому принадлежит аргумент. Программный код операторной функции указывается в фигурных скобках.

**Можно перегружать следующие операторы:** + - \* / % ^ & | ~ ! = < >  
+= -= \*= /= %= ^= &= |=  
<< >> >>= <<= == != <= >=  
&& || ++ -- ->\* , -> [] ()  
new new[] delete delete[]

**Нельзя перегружать операторы:**

1. :: - (разрешение области видимости)
2. . - (доступ к членам класса)
3. - (выбор члена через указатель на член)
4. - ? тернарный оператор

Прежде, чем рассматривать пример программы, рассмотрим указатель **this**, который используется в нем. Каждый метод внутри класса всегда имеет один скрытый аргумент, который является указателем и называется this. В метод также передается адрес того объекта, который вызывает этот метод. Указатель this сделали указателем на этот объект. Указатель this используют для упрощения кода программы, так как через него любой метод класса работает с объектом напрямую (не создавая локальной копии внутри функции) и дает возможность работы со всей конструкцией класса.

**Перегрузка бинарных и унарных операторов**

**Унарные операторы** – это операторы, которые имеют один операнд. Например: унарный плюс ++ (инкремент) или унарный минус – (декремент).

**Бинарные операторы** – это арифметические операторы сложения, вычитания и т.д.

**Правило 1:** При *перегрузке унарных операторов* функции аргументы не передаются за исключением перегрузки префиксной формы операторов инкремента и декремента, где в скобках () стоит признак для компилятора постфиксной формы оператора инкремента в виде типа поля объекта, for: (**int**). Причина связана с тем, что один аргумент операторной функции – объект, из которого вызывается эта функция. Доступ к этому объекту можно получить через указатель **this**.

**Правило 2:** При *перегрузке бинарных операторов* функции передается только один правый от знака операции аргумент. Левый аргумент в операторе сложения объектов отождествляется с объектом, из которого вызывается операторная функция, поэтому он не включается в список аргументов ОФ.

*Обобщим изложенное*: перегруженной операции всегда требуется количество аргументов на один меньше, чем количество операндов, так как один из операндов является объектом, вызывающим операторную функцию. Поэтому для унарных операторов аргументы не нужны. А для бинарных нужен один аргумент.

Разберем программу.Есть класс Cars (автомобили). В нем определено одно закрытое поле count – количество машин и 3 открытых метода: *конструктор по умолчанию дл*я инициализации поля count нулем, конструктор для инициализации поля count каким-то значением, отличным от нуля, и метод отображения информации на консоли. Задача состоит в том, чтобы создать два объекта и найти суммарное количество автомобилей, разность, инкременты количества в префиксной и постфиксной форме.

class Cars{

private:

int count;

public:

Cars():count(0) {} // инициализация нулем поле объекта

Cars(int addcount):count(addcount) {} // инициализация полей объектов **car1 и car2**

void Show(){cout<<"\n"<<count<<"\n";}

**/\***R-это обозначение правого операнда c**ar2**, доступ к нему через ссылку&, **Cars** – это его тип, левый операнд **car1** вызывает операторную функцию, поэтому его не включают в качестве аргумента операторной функции.\*/

**Cars operator + (const Cars&R){**

**/\*** В теле создание временного локального объекта **result** класса Cars, которому передается результат cars3*\*/*

**Cars result;**

**/\* З**начение поля объекта result.count (cars 3) = cars1+cars2, поле объекта result (cars 3) возвращается в качестве результата.

**result.count=count+R.count;**

return res;

}

// В префиксной версии аргументов нет

**Cars operator++(){**

**++count;**

//Указатель на объект (car3), для которого вычисляется оператор инкремента в main().

**return \*this;}**

/\*В постфиксной форме есть **int**, но это не аргумент, это сигнал для компилятора, что это постфиксная версия оператора\*/

**Cars operator++(int){**

**count++;**

**return \*this;}**

//Сложение объекта с числом

**Cars operator+( int &y){**

**Cars re;**

**re.count= count+y;**

**return re;}**

**}**

int main()

{

/\*создание 3-х объектов, для двух из них (car1 и car2) присваиваются значения полей через конструктор с одним аргументом, car3 определяется как сумма первых двух.\*/

**Cars car1(7), car2(3), car3=car1+car2, car4;**

// значение числа для сложения с объектом

int **y=1.5;**

**//** значение суммы полей выводится- на экран

**car3.Show();**

**/**/вычитание значений полей объектов

**car3=car1-car2;**

// значения разности объектов - на экран

**car3.Show();**

/\*команда префиксной формы оператора инкремента увеличивает значение поля объекта car3 на 1\*/

**++car3;**

//сумма объекта с числом

**car4= car3+y;**

**//** значение сумма объекта с числом – на экран

**car4.Show();**

return 0; }

Мы уже говорили, что перегруженные операции позволяют *вид программного кода сделать более читабельным, компактным и понятным****.***

Например, код операции сложения объектов в виде в точечном формате:

Cars.car3= Cars.car2+ Cars.car2;

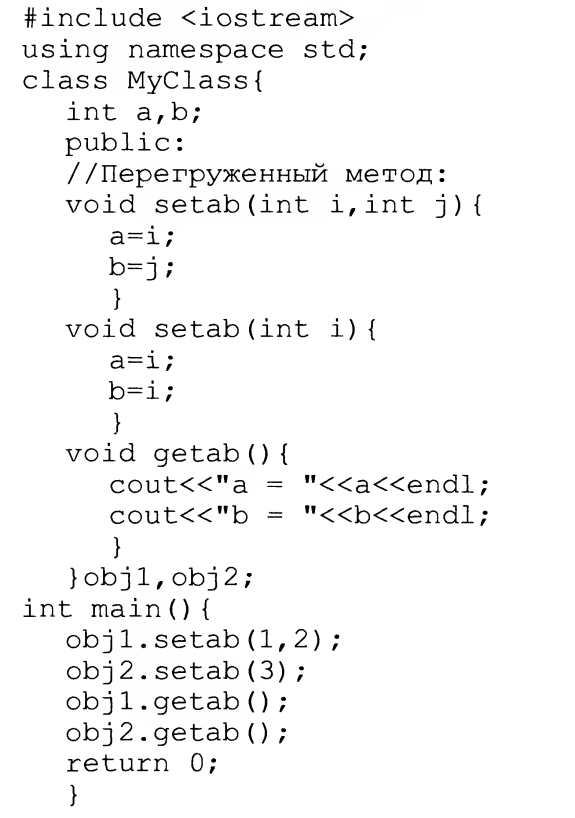
Операторная функция дает возможность заменить этот код на более простой:

car3= car2+ car2;

**Перегрузка методов**

Как и обычные функции, методы классов можно перегружать. В этом случае создается несколько вариантов одного и того же метода, но с разными прототипами. Отличие может быть связано с разными типами возвращаемых результатов или с разным типом и количеством аргументов. Такой подход позволяет использовать единый интерфейс и при этом учесть особенности вызова соответствующего метода с разным набором аргументов. Напомним, что данная концепция получила название полиморфизма.

Пример перегрузки методов класса:



А программе создан класс MyClass, в котором два закрытых целочисленных поля а и b, метод getab() для отображения значения закрытых полей и перегруженный метод для записи значений в закрытые поля. У метода setab() два варианта: с одним целочисленным аргументом и с двумя аргументами. Если методу передано два аргумента, то эти аргументы присваиваются в качестве значений полям а и b. Если у метода аргумент один, то соответствующее значение присваивается каждому полю а и b. Благодаря этому, в главном меню программы можно использовать как команду obj1.setab(1,2), так и команда obj2.setab(3). В зависимости от количества переданных методу setab() аргументов вызываются разные версии этого метода. Результат выполнения программы:

а=1

b=2

a=3

b=3

**Дополнительное задание**

1. Разработайте спецификации требований к ПО, которые решают эти задачи.
2. Спроектируйте ПО: входной и выходной интерфейсы для консоли, блок-схему программ.
3. Определитесь со структурой программы (прототипы, описание, вызов функций), разработайте код на языке объектно-ориентированного программирования С++, протестируйте и отладьте его.

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №5**

1. Что такое полиморфизм?
2. Что такое перегрузка операторов?
3. Что такое переопределение операторов?
4. Каков синтаксис операторной функции?
5. Какие операторы можно перегружать, а какие нельзя?
6. Какова зависимость между количеством аргументов в операторной функции и количеством операндов? Объясните эту зависимость.
7. Каким образом перегруженные операции позволяют вид программного кода сделать более читабельным?

**Лабораторная работа №6**

**Тема: Наследование**

**Задания:**

1. Проблему экономии времени, доходов, а также увеличения надежности создаваемой объектно-ориентированной программы можно решить с помощью одного из главных принципов ООП - наследования, используя для создания класса-потомка готовый протестированный и отлаженный код класса-родителя. Для решения этой проблемы используйте преимущества **простого наследования**. Для этого создайте родительский класс работник (**Employee)**, протестируйте и отладьте его. В описании этого класса есть поля**:** имя работника - занимаемая должность - position[64], заработная плата за месяц- salary;

методы: конструктор по умолчанию, конструктор с аргументами, деструктор, расчет заработной платы за год, размещение информации о классе-родителе на консоли.

На основе отлаженного класса-родителя **Employee** создайте с помощью **механизма наследования** **public** класс-потомок менеджер (**Manager)**.

Класс - потомок Manager состоит из**:**

полей**:** ежегодный бонус **(annual\_bonus)**, машина компании (**company\_car[64]**)**,** опционы на акции (**stock\_options**) типа (**int**);

методов: конструктор по умолчанию, конструктор с аргументами, деструктор, размещение информации о классе-потомке на консоли.

*Определитесь с идентификаторами доступа к членам класса, не нарушая основного принципа ООП – инкапсуляции в наследовании.*

2. Используйте преимущества **множественного наследования,** а именно: разработайте, протестируйте и отладьте родительские классы Дисплей **(*Display*)** и Материнская плата (M***otherBoard),*** и на их основе создайте класс-потомокКомпьютер **(*Сomputer*)**:

В описании *класса-родителя* ***Display*** есть

поля: тип монитора char type[32], количество цветов long colors, разрешение по оси х int x\_resolution, разрешение по оси у int y\_resolution.

методы: конструктор с аргументами; размещение информации о классе-родителе на консоли.

В описании *класса-родителя* ***MotherBoard*** *имеет*

поля: тип процессора int processor, скорость процессора int speed, объем оперативной памяти int RAM;

методы: конструктор с аргументами; размещение информации о классе-родителе на консоли.

- *Класс-потомок* ***Сomputer*** создать на основе родительских классовс помощью механизма наследования **public** и отобразить его на экране**,** добавив:

поля: марка компьютера (name [64]), объем жесткого диска (hard\_disk**)**.

методы: конструктор с аргументами, метод Show().

3. В задании № 1 изменить механизм наследования так, чтобы закрыть доступ к членам базового класса из производного класса. А затем восстановить этот доступ всеми известным Вам способами.

**Дополнительное задание**

1. Разработайте спецификации требований к ПО, которые решают эти задачи.

1. Спроектируйте ПО: входной и выходной интерфейсы для консоли, блок-схему программ.
2. Определитесь со структурой программы (прототипы, описание, вызов функций), разработайте код на языке объектно-ориентированного программирования С++, протестируйте и отладьте его.

**Методические указания**

Для выполнения лабораторной работы № 5 требуется знать основные понятия: *наследование, базовый (родительский) класс, класс-потомок (производный класс), идентификатор доступа, механизм наследования, простое наследование, множественное наследование, многократное наследование.*

**Наследование —** механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса.

**Класс-потомок** может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами.

Наследование позволяет строить **иерархии классов.**

Наследование - важная часть ООП. Выигрыш от него состоит в том, что наследование позволяет использовать существующий код несколько раз. Имея написанный и отлаженный базовый класс, мы можем его больше не модифицировать, при этом механизм наследования позволит нам приспособить его для работы в различных ситуациях. Используя уже написанный код, мы экономим время и деньги, а также увеличиваем надежность программы.

При наследовании классов производный класс создается не на пустом месте, а на основе базового класса, получая в наследство от своего родителя поля и методы. Тем не менее из того, что производный класс наследует свойства базового класса, еще не следует, что он наследует *все* свойства. Какие поля и методы наследуются, а какие нет, определяется доступностью соответствующих членов базового класса и *механизмом наследования*.

Для объявления открытых членов класса используют идентификатор public. Закрытые члены класса объявляются с *идентификатором доступа* private (или без идентификатора доступа). Кроме того, члены класса могут объявляться с *идентификатором доступа* protected. В этом случае член класса называется защищенным. Такой член класса является закрытым, а от private-члена он отличается способом наследования.

Ключевые слова public, private, protected используются не только для определения доступа к членам класса, но и являются *индикаторами механизма наследования*.

Таким образом, в зависимости от типа наследования (public, private, protected) и доступности членов (public, private, protected) получаем разный результат.

Например, public-член базового класса наследуется производным классом при любом механизме наследования, но в производном классе унаследованный член имеет разный уровень доступа. При public-наследовании он останется public-членом в производном классе.

При private-наследовании public-член базового класса становится private-членом

производного класса. Существует 9 вариантов наследования. Все они перечислены в таблице 1.

**Таблица 1**

|  |  |  |  |
| --- | --- | --- | --- |
| **Механизм** | public- наследование | private- наследование | protected - наследование |
| **Доступность** |  |  |  |
| рublic - член | public | private | protected |
| рrivate- член | нет доступа | нет доступа | нет доступа |
| рrotected - член | public | private | protected |

Информацию о наследовании, представленную в виде таблицы 1, можно сформулировать в виде нескольких правил, которые приведены в порядке убывания важности:

1. private-члены базового класса не наследуются;
2. при public-наследовании уровень доступа члена класса не меняется;
3. при private-наследовании наследуемые члены становятся private-членом производного класса;
4. при protected-наследовании наследуемые члены становятся protected -членами производного класса.

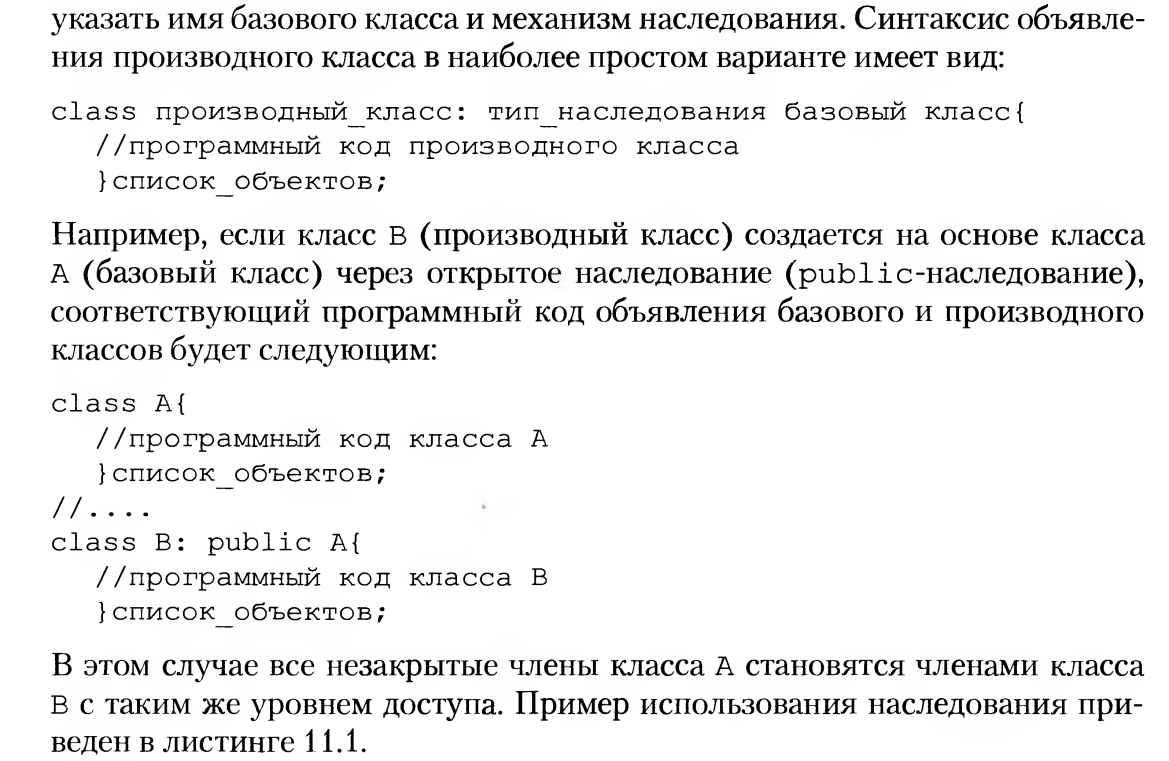
**Синтаксис** объявления производного класса при наследовании базового класса имеет вид:

Class *производный\_класс*: тип\_наследования *базовый\_класс* {

программный код производного класса

};

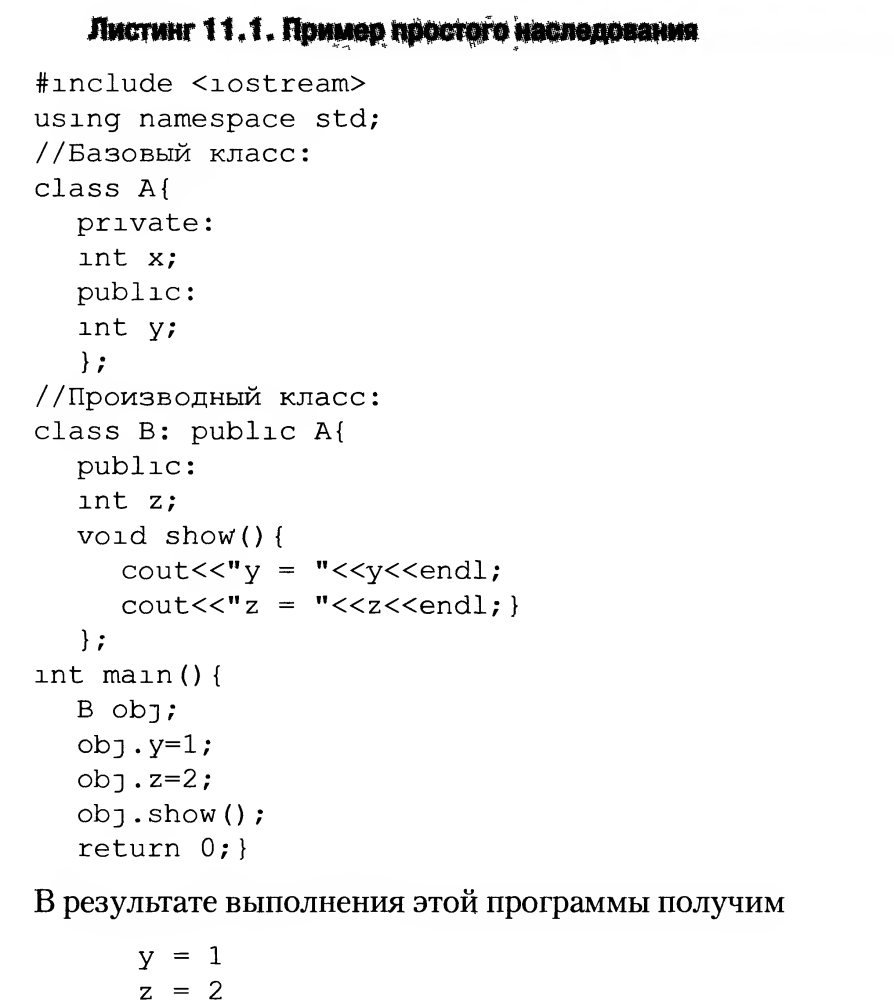
Например, если класс В (производный класс) создается на основе класса А (базовый класс) через открытое наследование (public-наследование), соответствующий программный код объявления базового и производного классов будет следующим:



Наследование бывает *простое, множественное и многократное.*

**Простое наследование -** это когда у производного класса имеется один базовый класс.

Пример простого наследования.

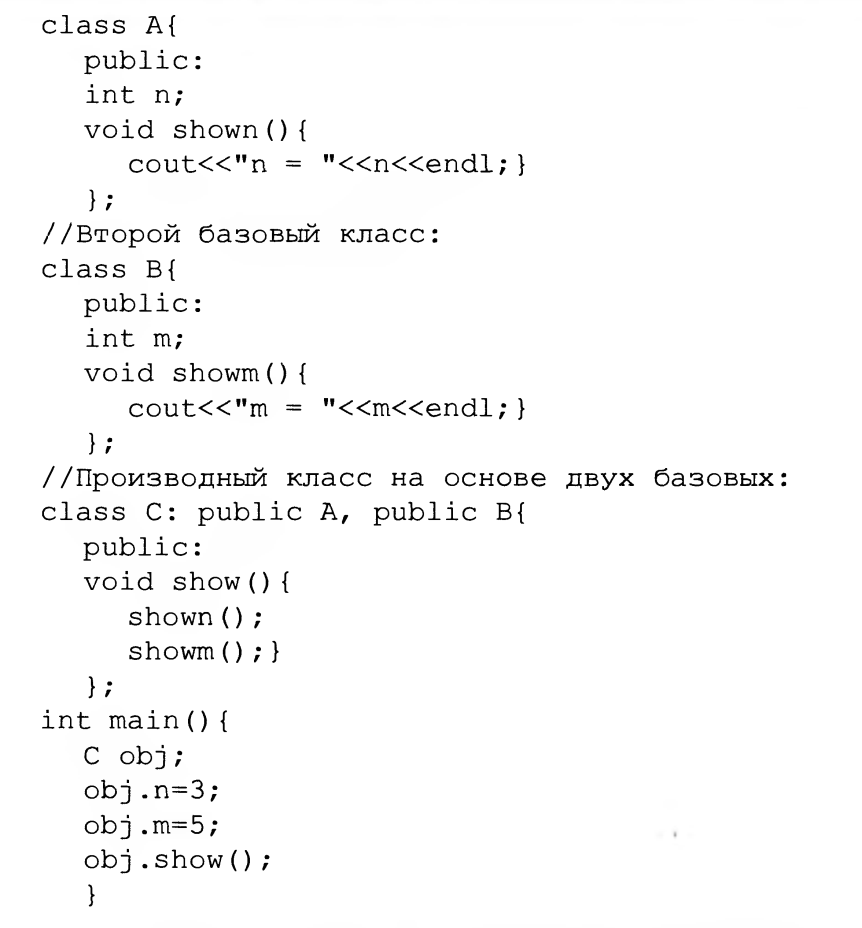


Здесь наследуется поле **у –** оно открытое для класса В благодаря **public –** наследованию. Не наследуется поле **х** – оно закрытое, так как не имеет доступа к члена типа **private**. Класс- потомок **В** использует еще и свое поле z и свой метод **show**.

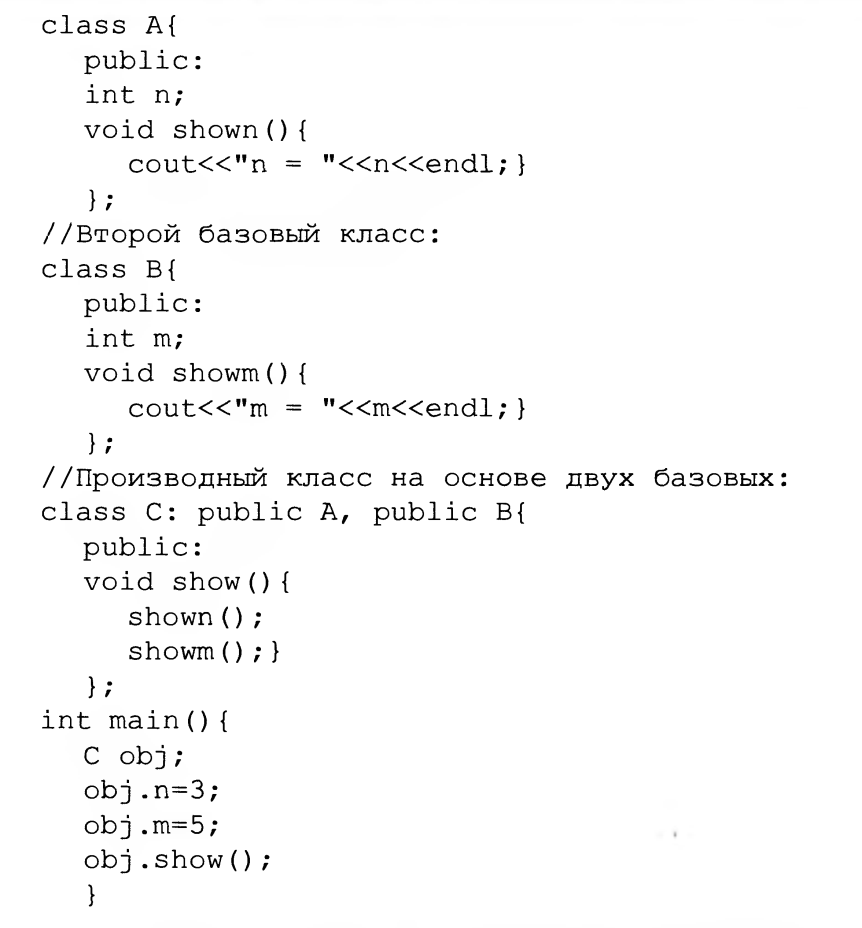
**Множественное наследование -** когда у производного класса имеется несколько базовых классов. **Многократное наследование –** это когда производный класс может иметь и базовый и производный классы.

Рассмотрим пример множественного наследования:

//Первый базовый класс А



//Второй базовый класс В

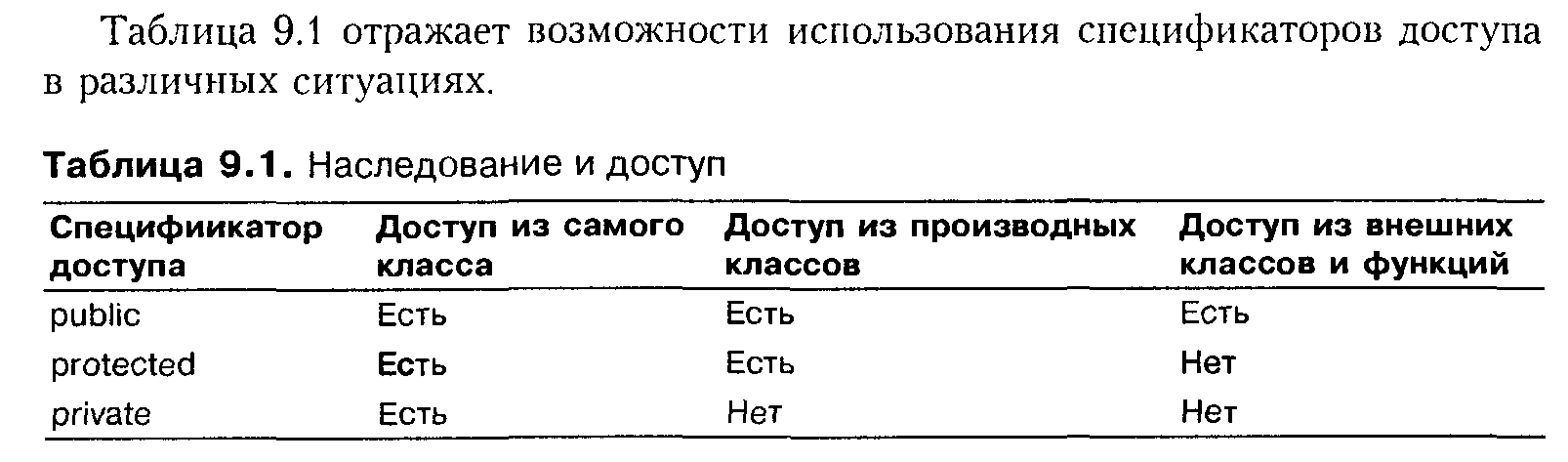


Здесь рассматриваем самый простой случай, когда все поля открыты и используется public –наследование. Класс **С** наследует поле **n** и метод **shown ()** из класса **А**, поле **m** и метод **showm ()** из класса **В**. То есть класс С наследует все поля и методы классов Аи В.

Если заменить механизмы public наследования на private A и protected B, в функции main() команды obj.n=3 и obj.m=5 и метод obj.show() отметятся красным, как ошибочные, так как доступ к ним утерян из-за замены механизма наследования.

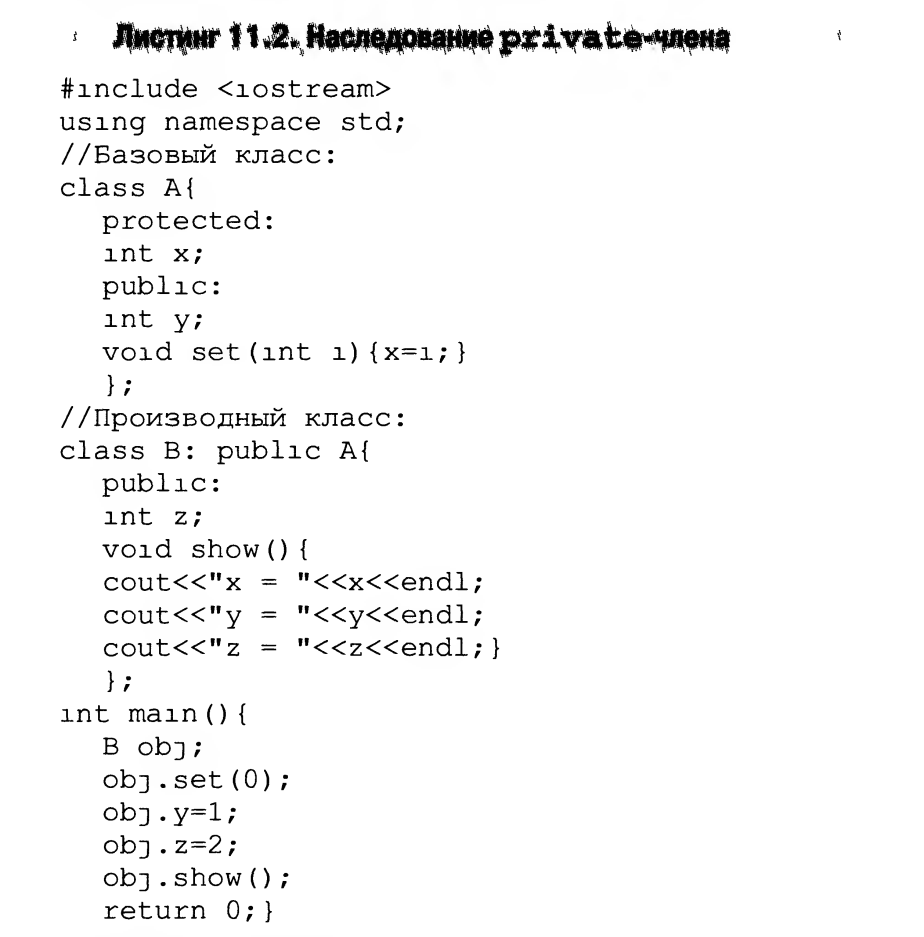
Рассмотрим таблицу 2 использования спецификаторов доступа в различных ситуациях, а именно: при доступе к членам базового класса из самого класса, из производных классов и из внешних классов и функций.

**Таблица 2**



Из таблицы можно сделать вывод, что если Вы пишете класс, который будет использоваться при наследовании, как базовый, то данные, к которым должен быть доступ объявите, как **protected**.

Для примера этого положения внесем изменения в программный код с простым наследованием. В описании класса А поле х объявим, как **protected**. Поэтому поле х остается закрытым. Но теперь поле х наследуется классом В. У объектов класса В теперь будет, кроме полей y и z, еще и поле x. Правда, как и в случае класса А, к полю х объектов класса В доступ извне невозможен. Поэтому для проверки функциональности поля х добавим вспомогательный программный код для метода set (), устанавливающего значение полю х (этот метод поместим в класс А, а в классе В он появится через механизм наследования), и внесем небольшие изменения в метод show().



Для инициализации значения поля х объекта obj класса B в главном методе программы вызывается метод set (), который наследуется из класса А.

Как отмечалось, напрямую обратиться к полю х в данном случае возможности нет – хотя поле и наследуется классом В, оно является защищенным (т.е. закрытым). Тем не менее, внутри класса В обращение к полю является корректным, например, как при описании метода show (). Результат выполнения программы:

х=0

у=1

z=2

Если теперь изменить механизм наследования, например, на private-наследование, класс В будет иметь те же поля и методы, но теперь поля х и y и метод set () будут закрытыми. К этим членам класса нельзя будет обратиться вне класса.

Как восстановить уровень доступа унаследованного члена класса.

При **private** наследовании красным отметятся команды **obj.set(0)** и **y=1**, как ошибочные, так как доступ к ним закрылся. Для **private** наследования рассмотрим два способа восстановления доступа к закрывшимся полям **y** и методу **set**.

**1 способ**. Создадим новый метод **setb** в класса **В**, чтобы осуществить доступ к закрытым полям **х** и **у** класса **А**.

#include <iostream>

using namespace std;

class A{

protected:

int x;

public:

int y;

void set(int i) {x=i;}

};

//proizv klass

class B: private A{

public:

int z;

void setb (int j,int f){x=j, y=f;}

void show() {

cout<<"x = "<<x<<endl;

cout<<"y = "<<y<<endl;

cout<<"z = "<<z<<endl;

}};

int main(){

B obj;

obj.setb(5,10);

obj.z=2;

obj.show();

return 0;

}

Результат работы программы: х=5, у=10, z=2

2 способ. **Восстановление уровня доступа**

#include <iostream>

using namespace std;

class A{

protected:

int x;

public:

int y;

void set(int i) {x=i;}

};

//proizv klass

class B: private A{

public:

int z;

A::y; //восстановление уровня доступа к полю

A::set; // восстановление уровня доступа к методу

void show() {

cout<<"x = "<<x<<endl;

cout<<"y = "<<y<<endl;

cout<<"z = "<<z<<endl;

}};

int main(){

B obj;

obj.set(0);

obj.y=15;

obj.z=25;

obj.show();

return 0;

}

Результат работы: x=0, y=15, z=25

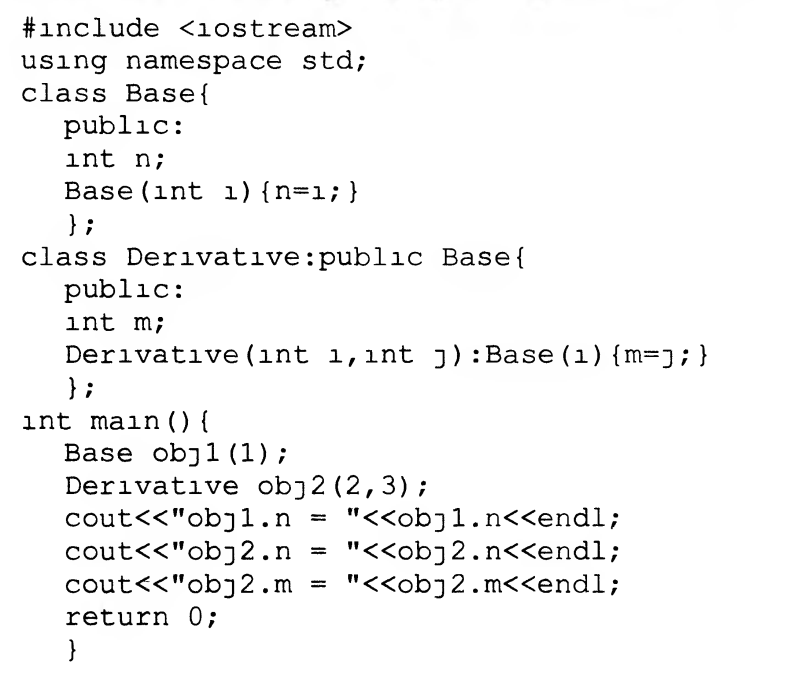
Этот способ использовать в крайних случаях, т.к. он нарушает принцип инкапсуляции.

**Простой конструктор при наследовании**

Важным и принципиальным является вопрос использования конструкторов и деструкторов при наследовании. Действительно, если в базовом классе имеется конструктор с аргументами, то в производном классе как минимум должен быть предусмотрен механизм передачи аргументов этому конструктору. Такой механизм существует. С точки зрения синтаксис языка С++ сводится он к тому, что при определении конструктора производного класса после имени конструктора указывается имя конструктора базового класса с указанием в круглых скобках аргументов, передаваемых этому конструктору. Если базовых классов несколько, для каждого из них через запятую перечисляются конструкции вида *имя\_конструктора (аргументы)* для передачи аргументов конструктору соответствующего базового класса. Синтаксис объявления конструктора производного класса имеет вид:

**имя\_конструктора производного класса (аргументы конструктора базового класса, аргументы конструктора производного класса): имя\_ конструктора базового класса (аргументы базового класс), {имя\_атрибута1 производного класса = значение; …}**

Рассмотрим листинг программы:

****

В базовом классе Base объявлен один конструктор, предусматривающий передачу аргумента.Этимконструктором инициализируется значение целочисленного поля n класса. При создании производного класса Derivative необходимо предусмотреть передачу аргумента конструктору базового класса. Причина – в способе создания объектов производных классов. Дело в том, что при создании объектов производного класса сначала вызывается конструктор базового класса, а уже затем конструктор класса производного. Если базовых классов несколько, то их конструкторы вызываются в порядке наследования классов (очередность в списке наследуемых классов слева направо). В конструкторе производного класса Derivative предусмотрена передача двух аргументов: один передается конструктору базового класса, а значение второго аргумента используется для инициализации целочисленного поля m, описанного в производном классе. В главном методе программы создается объект базового класса и объект производного класса. В первом случае конструктору передается один аргумент, во втором – два. В результате выполнения программы получаем:

obj1.n = 1

obj2.n = 2

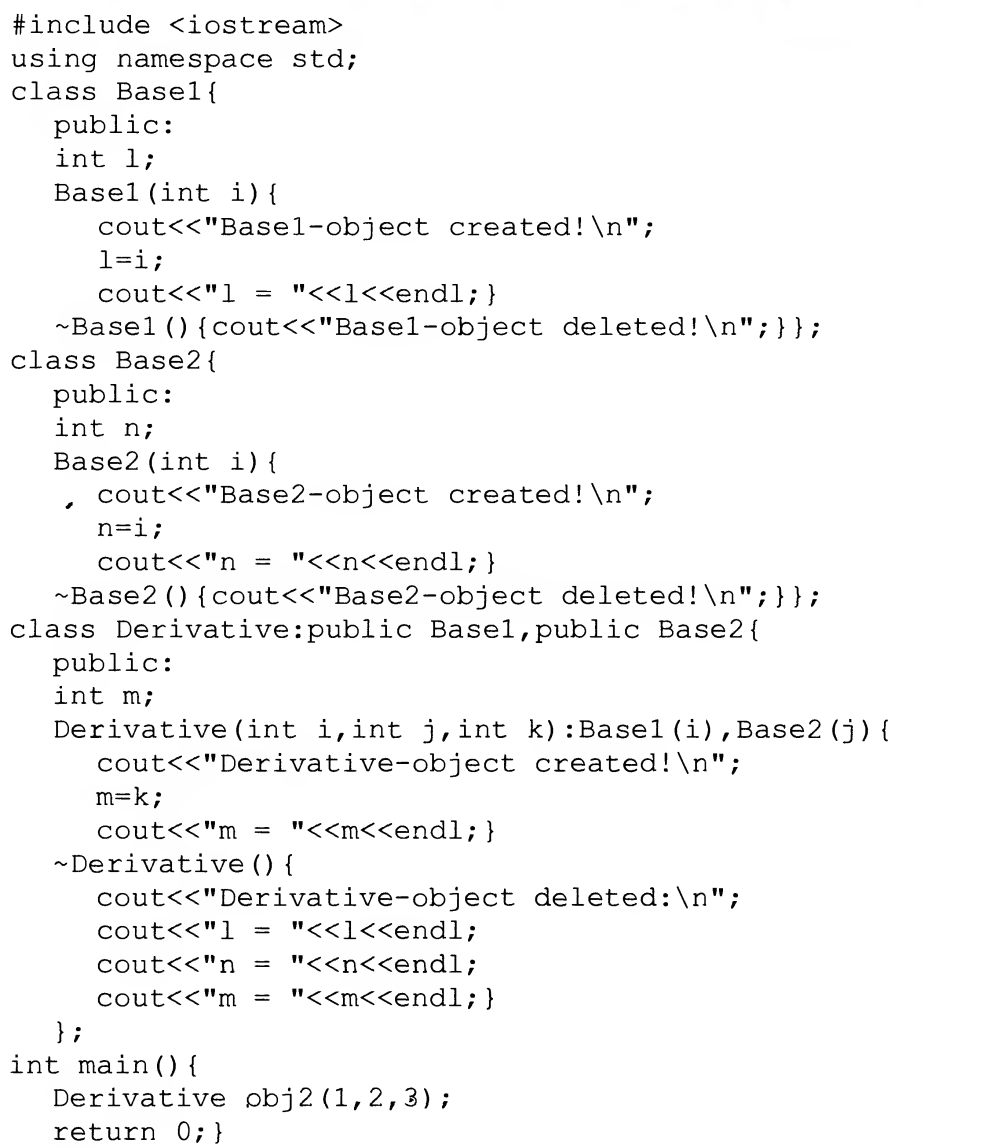
obj2.m =3

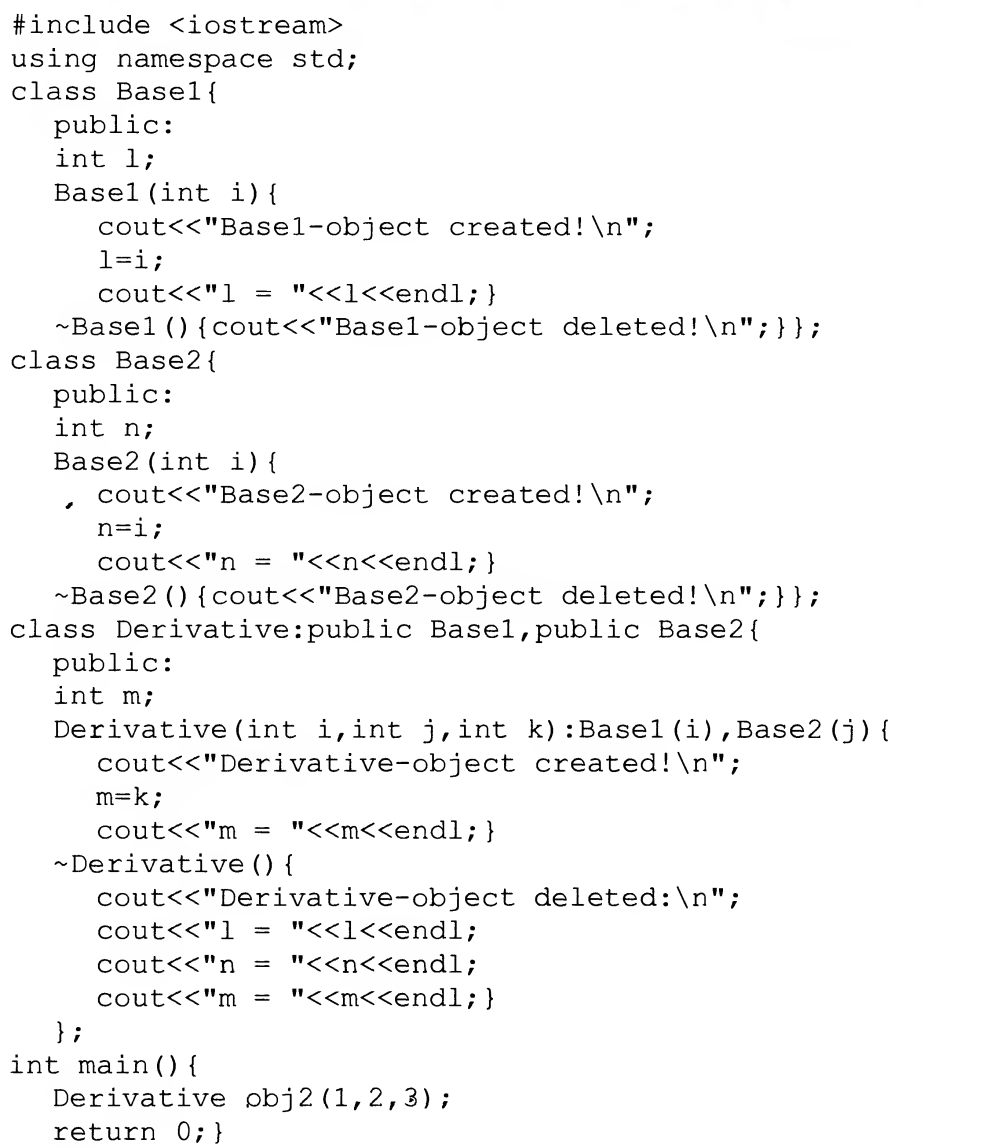
**Конструктор при множественном наследовании**

**Синтаксис конструктора**

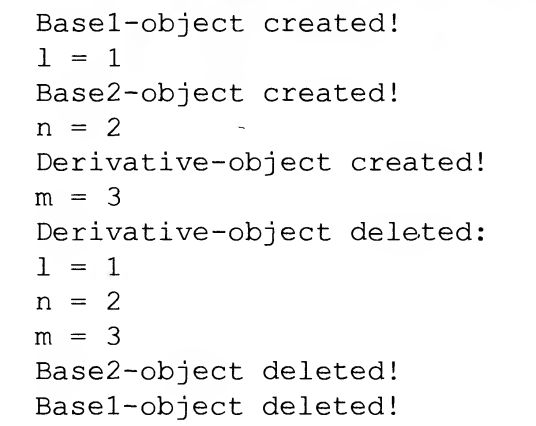
**Имя\_конструктора производного класса (аргументы конструктора базового класса, аргументы конструктора производного класса): имя\_ конструктора базового класса 1 (аргументы базового класс 1), имя\_ конструктора базового класса 2(аргументы базового класс 2),... {имя\_атрибута1 производного класса = значение; …}**

Пример определения конструктора производного класса при многократном наследовании приведен в листинге:





Этот же пример иллюстрирует вызов деструкторов. Правило вызова деструкторов состоит в том, что деструкторы вызываются в обратном порядке, если сравнить с последовательностью вызова конструкторов. Первым вызывается деструктор производного клааса, а после этого деструкторы базовых классов (очередность вызова деструкторов – справа налево в списке наследования базовых классов). Для удобства при вызове конструкторов и деструкторов базовых и производных классов выводятся следующие сообщения. Главный метод программы состоит всего из одной команды создания объекта производного класса. В результате выполнения программы получаем:



Вся работа по выводу сообщения выполняется и деструкторами, которых в данном случае по три (конструктор и деструктор для каждого из двух базовых классов и одного производного). Конструкторы вызываются в такой последовательности: конструктор класса Base1, конструктор класса Base2 и затем конструктор класса Derivative. Деструкторы вызываются в обратной последовательности: деструктор класса Derivative, деструктор класса Base2, и, наконец, деструктор класса Base1.

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №6**

1. Что такое в ООП? Приведите примеры.
2. В чем заключается выигрыш для программиста при использовании им наследования?
3. Какие члены базового класса не наследуются?
4. От чего зависит доступ к членам базового класса из произвольного класса?
5. Чем отличается идентификатор доступа от механизма доступа?
6. Назовите основные правила наследования (используйте таблицу наследования).
7. Синтаксис объявления производного класса, как наследника базового класса?
8. Что такое простое, множественное и многократное наследование?
9. Особенности использования идентификатор доступа protected при наследовании классов?
10. Какие существуют виды конструкторов при наследовании?
11. Каков синтаксис объявления конструктора производного класса?
12. Каковы особенности конструктора при простом наследовании?
13. Каковы особенности конструктора при многократном наследовании?
14. Какова последовательность вызова конструкторов и деструкторов при наследовании?

**Лабораторная работа №7**

**Тема: Виртуальные методы**

**Задания:**

1. Создать базовый класс Array с полями: массив типа unsigned и поле для хранения количества элементов у текущего объекта массива. Максимально возможный размер массива задается статической константой. Реализовать конструктор инициализации, задающий количество элементов и начальное значение (по умолчанию 0). Реализовать в классе Array виртуальную функцию поэлементного сложения массивов. Реализовать два класса, переопределив виртуальную функцию сложения. Вызывающая программа должна продемонстрировать все варианты вызова виртуальных функций.
2. *\*Создать абстрактный базовый класс Figure с виртуальными методами вычисления площади и периметра. Создать производные класса: Rectangle (прямоугольник), Circle (круг), Trapezium (трапеция) со своими функциями площади и периметра. Самостоятельно определить, какие поля необходимы, какие из них можно задать в базовом классе, а какие в производных. Площадь трапеции: *S = (a + b) x h/2.*
3. Создать абстрактный базовый класс Currency (валюта) для работы с денежными суммами. Определить виртуальные функции перевода в рубли и вывода на экран. Реализовать производные классы Dollar (доллар) и Euro (евро) со своими функциями перевода и вывода на экран.

**Дополнительное задание**

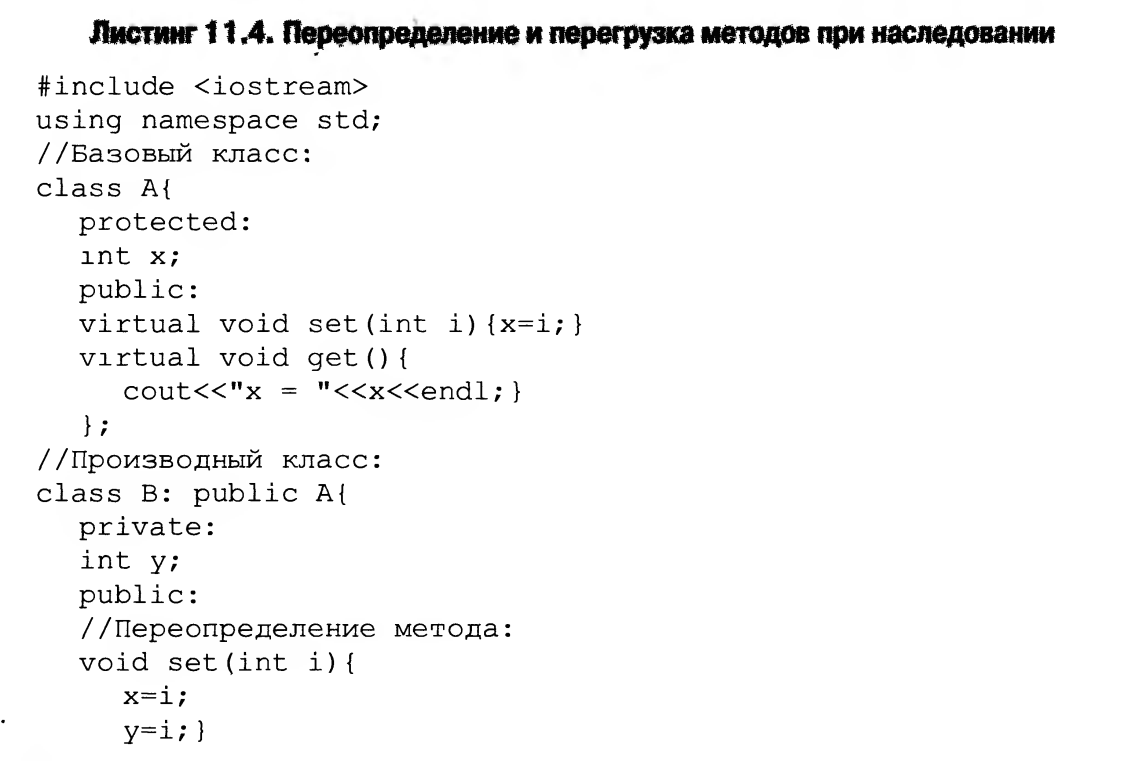
1. Разработайте спецификации требований к ПО, которые решают эти задачи.
2. Спроектируйте ПО: входной и выходной интерфейсы для консоли, блок-схему программ.
3. Определитесь со структурой программы (прототипы, описание, вызов функций), разработайте код на языке объектно-ориентированного программирования С++, протестируйте и отладьте его.

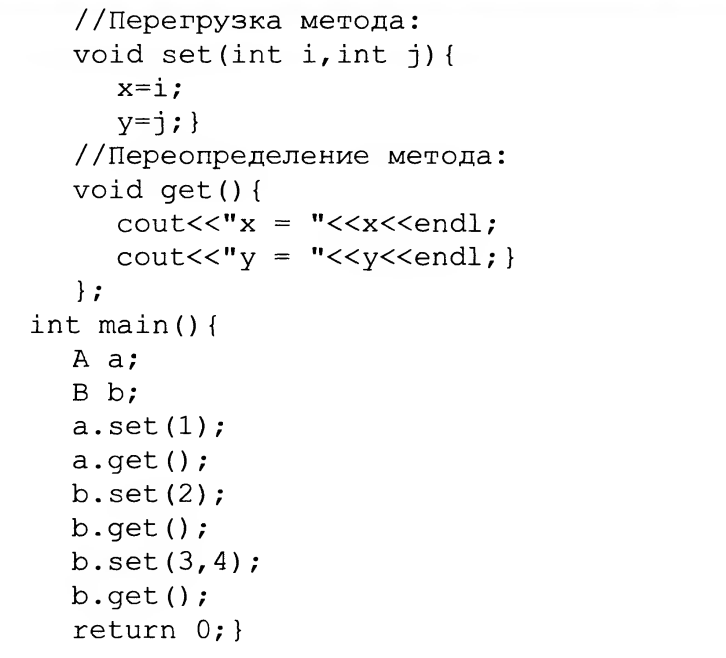
**Методические указания**

Для выполнения лабораторной работы № 6 требуется знать основные понятия: *виртуальный метод, переопределение метода при наследовании, перегрузка метода при наследовании, чисто виртуальная функция.*

**Виртуальный метод** - это метод, используется в базовом классе, который дает возможность переопределения методов в производном классе при наследовании.

Виртуальные методы описываются с помощью ключевого слова virtual в базовом классе. При наследовании виртуальный метод может быть переопределен и перегружен. При перегрузке меняется прототип метода. При переопределении прототип метода остается неизменным. Рассмотрим пример.





При объявлении методов set () и get () в классе А использовано ключевое слово virtual. Благодаря этому в классе В, производном от класса А, методы set () и get() переопределяются. Но метод set () еще и перегрузить можно. Метод set () в классе А описан с одним аргументом, значение которого присваивается полю Х.

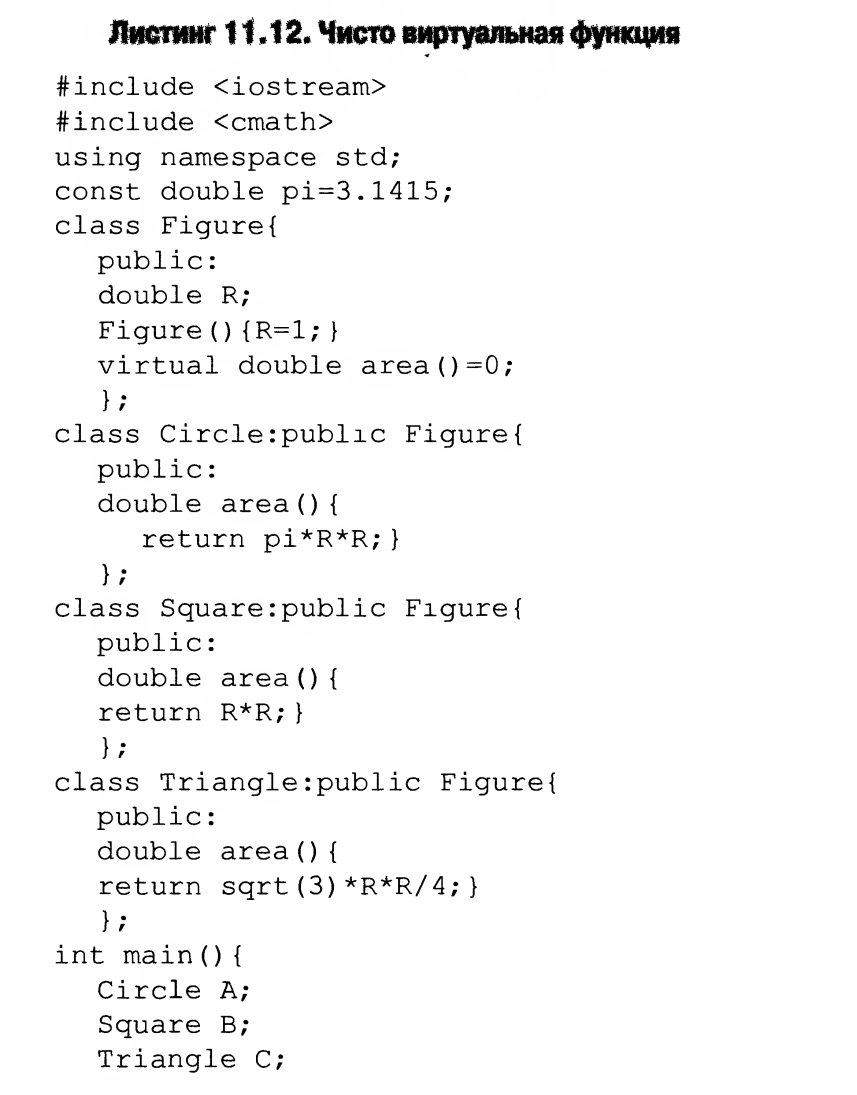
В классе В значения надо присвоить двум полям Х и У. Класс В содержит описание двух вариантов метода set () – с одним аргументом и с двумя. В том случае, если методу передается два значения аргумента из функции main (), эти аргументы определяют поля Х и У соответственно. В случае, когда методу передается значение одного аргумента, поля х и у получают одинаковые значения.

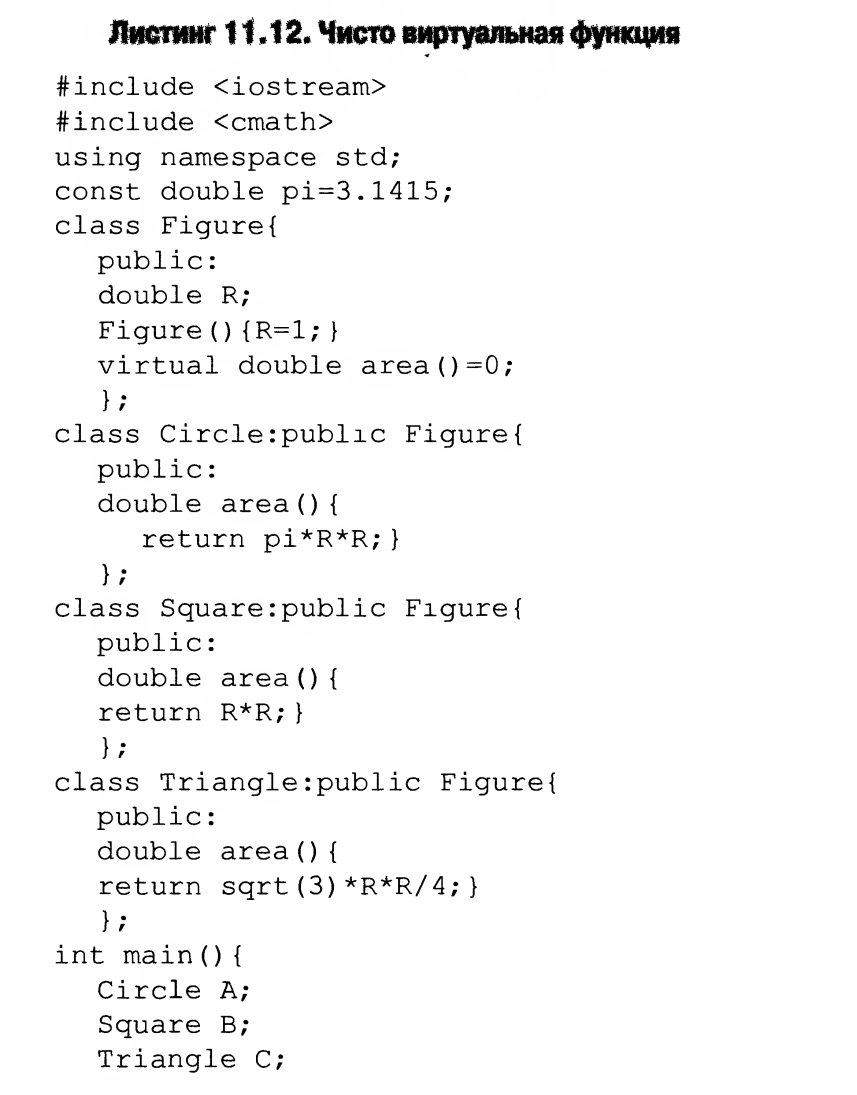
Прототип варианта метода set () с одним аргументом совпадает с прототипом этого метода, описанном в базовом классе. В этом случае речь идет о переопределении метода. Перегрузка метода set () осуществляется путем определения в производном классе варианта метода set()с двумя аргументами. Программный код, используемый при вызове метода, осуществляется на основе того, из какого объекта вызывается метод и какой у метода прототип.

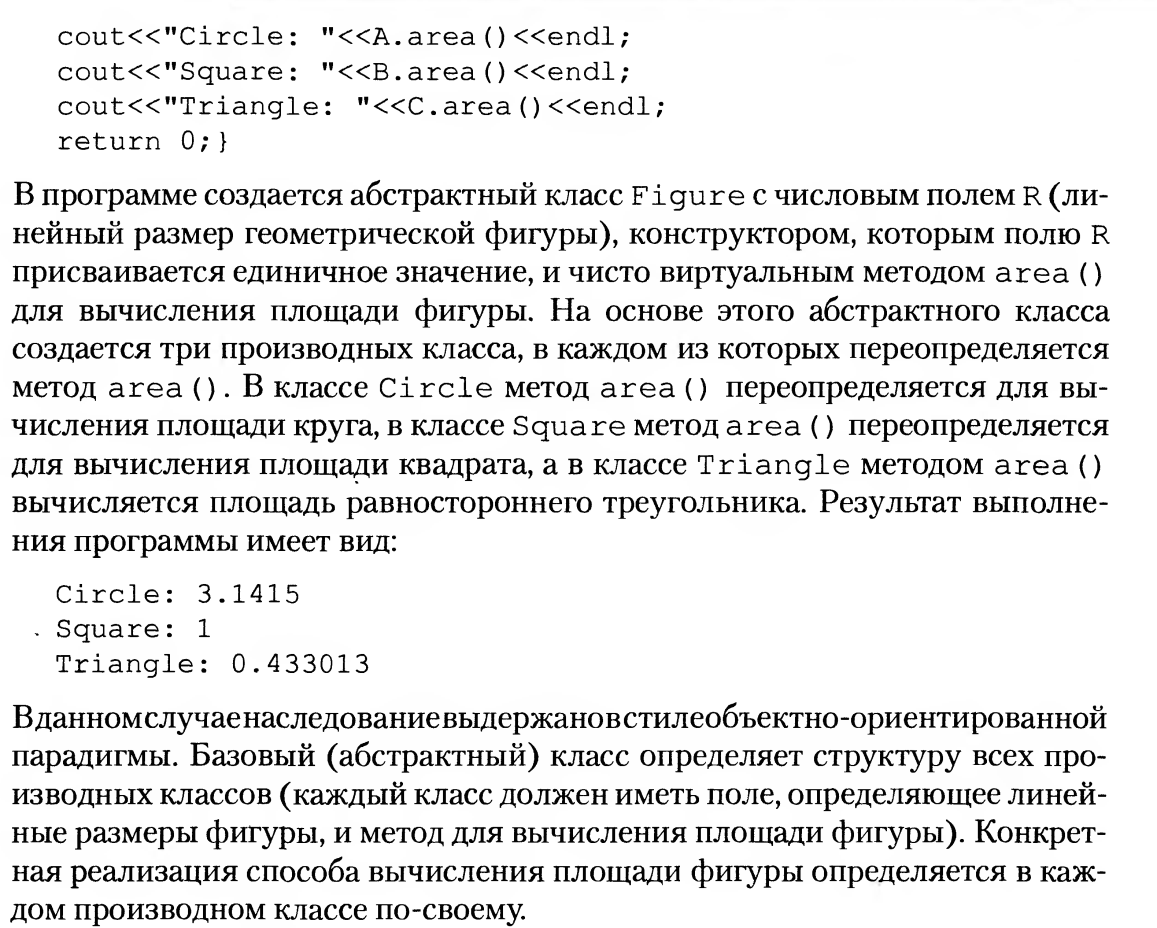
**Чисто виртуальной функцией** называется такая виртуальная функция, которая не имеет определения в базовом классе. Чтобы определить чисто виртуальную функцию, после прототипа функции необходимо указать оператор присваивания (знак равенства) и ноль. Для чисто виртуальной функции блок с кодом функции (тело) не указывается. Синтаксис чисто виртуальной функции имеет вид:

virtual тип\_результата имя-функции (аргументы)=0;

Класс, содержащий хотя бы одну чисто виртуальную функцию, называется абстрактным. В производном классе чисто виртуальная функция должна быть переопределена, поскольку из базового класса эта функция вызвана быть не может, там для неё нет программного кода.







В программе создается абстрактный класс **Figure** с числовым полем **R** (линейный размер геометрической фигуры), с конструктором, которым полю **R** присваивается единичное значение, и чисто виртуальный метод **area()** для вычисления площади фигуры. На основе этого абстрактного класса создадим 3 производных класса, в каждом из которых переопределяется метод area(). В классе **Circle** метод **area()** переопределяется для вычисления площади круга, в классе Square метод **area()** переопределяется для вычисления площади квадрата, а в классе Triangle метод **area()** переопределяется для вычисления площади равностороннего треугольника. Результат выполнения программы имеет вид:

circle: 3.1415

square: 1

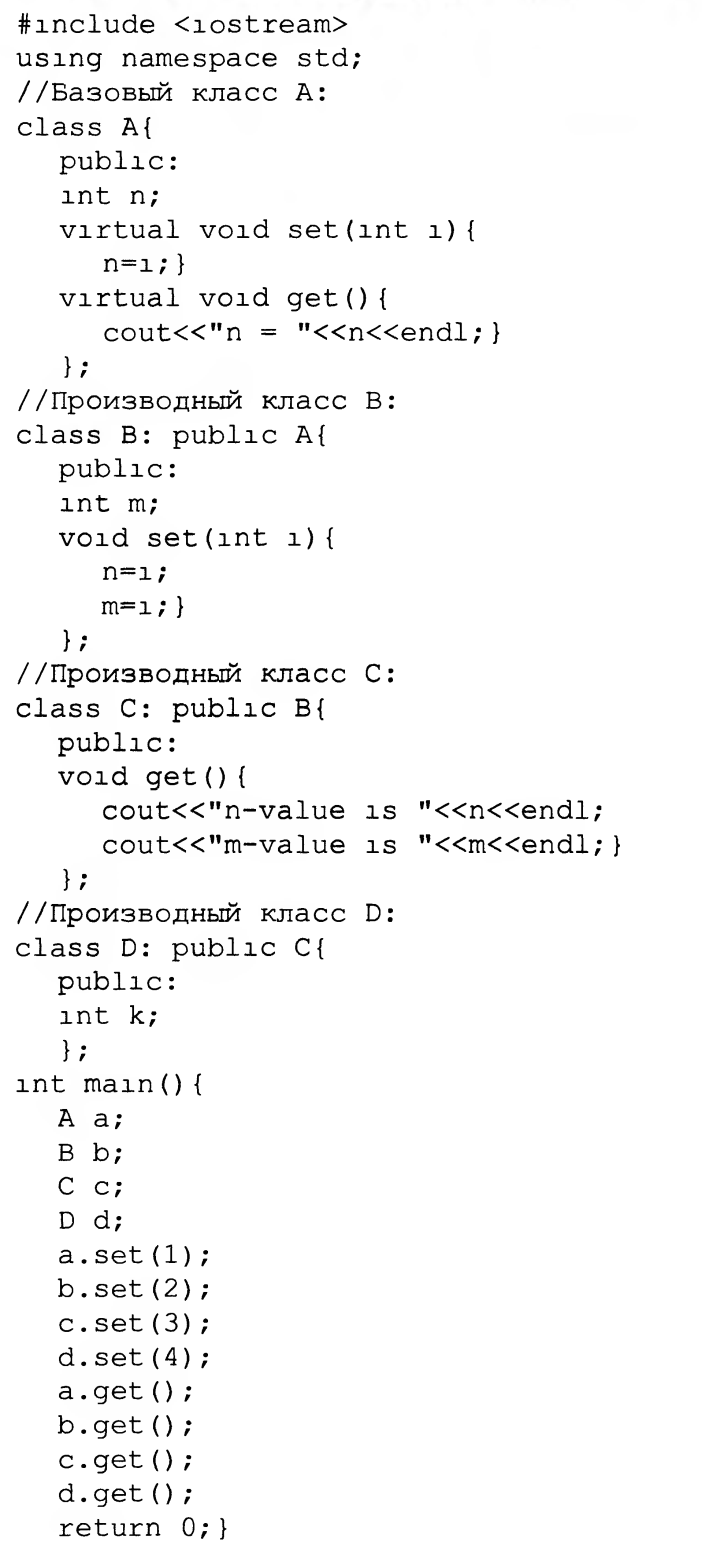
triangle: 0.433013

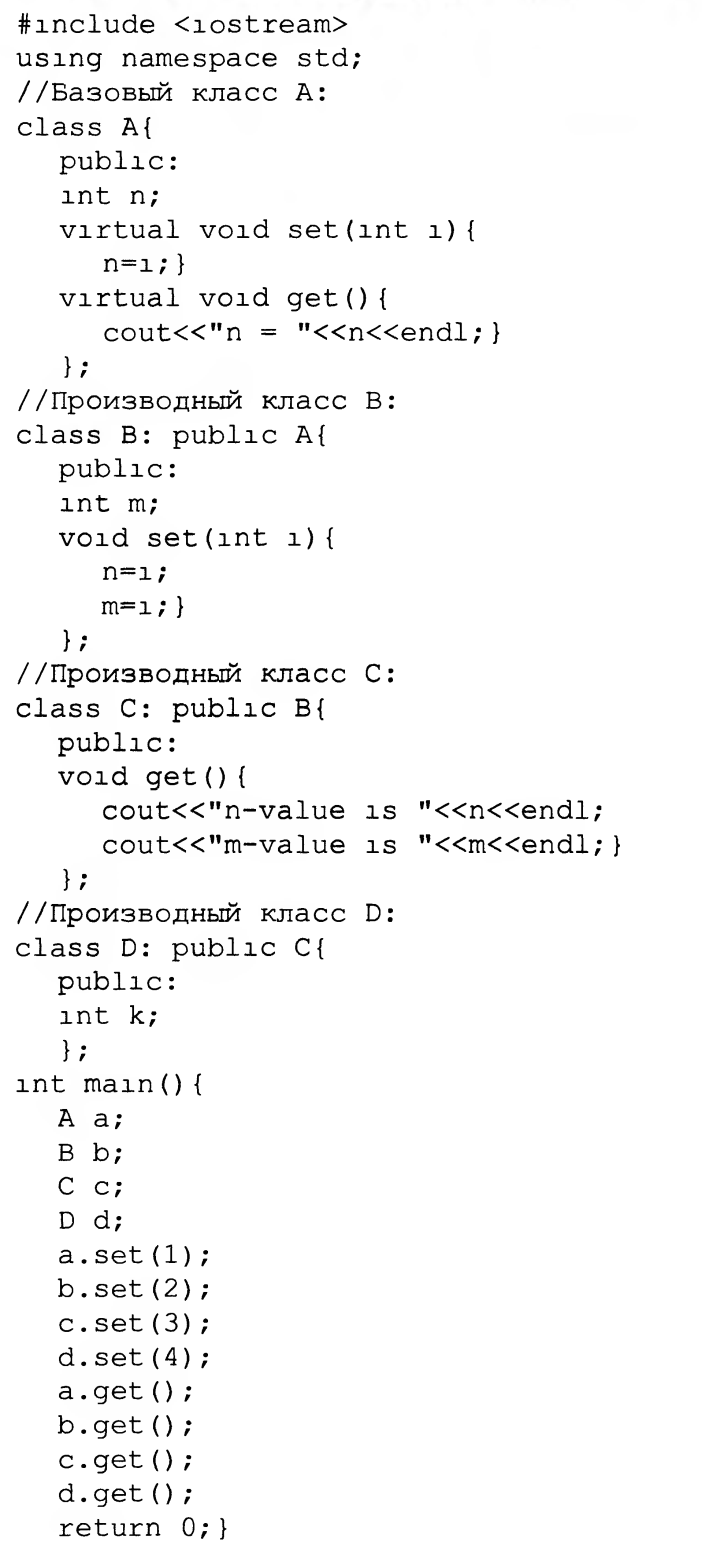
Базовый (абстрактный) класс определяет структуру всех производных классов (каждый класс должен иметь поле, определяющее линейные размеры фигуры, и метод для вычисления фигуры). Конкретная реализация способа вычисления площади фигуры определяется в каждом производном классе по-своему.

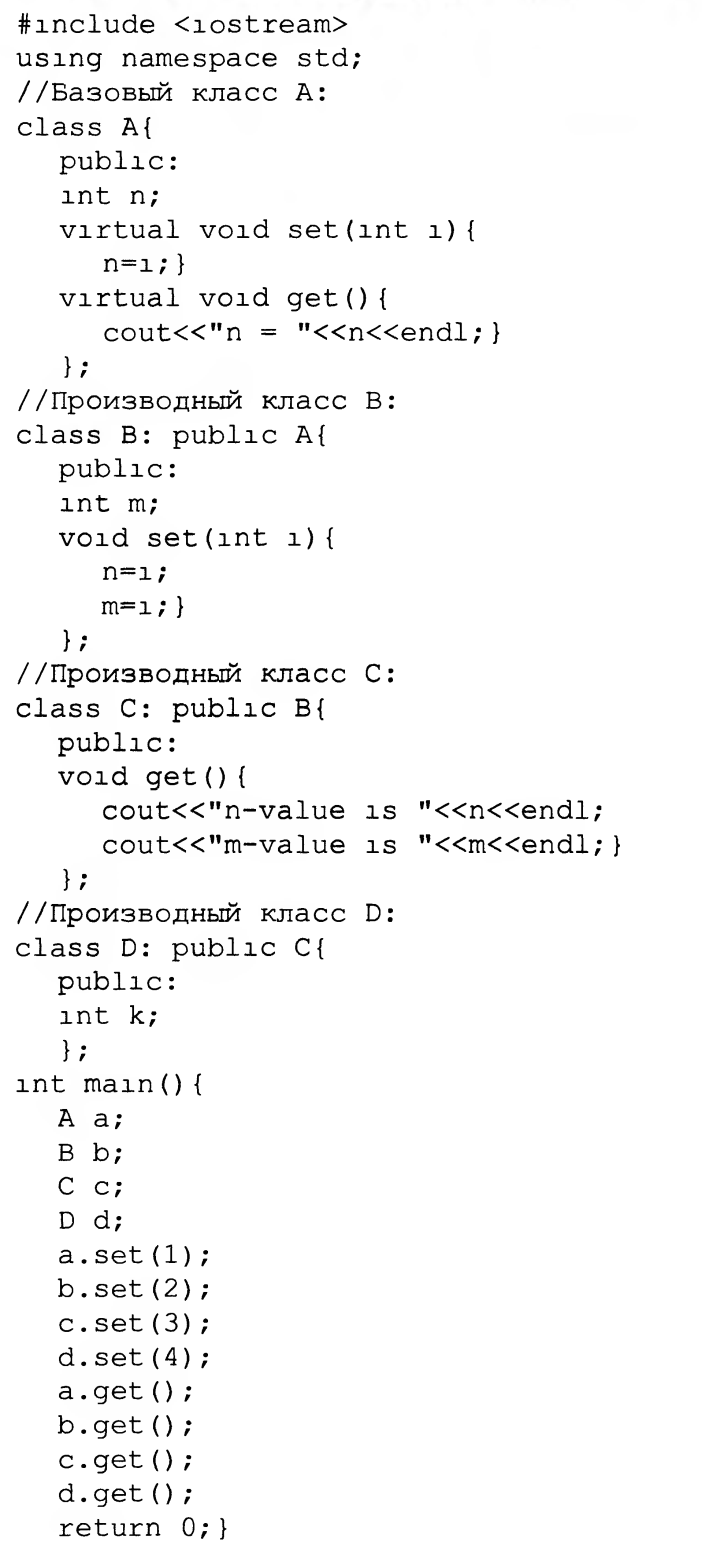
**Виртуальные методы в многоуровневом наследовании**

При многоуровневом наследовании, как отмечалось выше, производный класс, в свою очередь, является базовым для следующего класса, и так далее. Образуется иерархическая структура, в вершине которой находится базовый класс, на основе которого по цепочке последовательного наследования создаются новые классы.

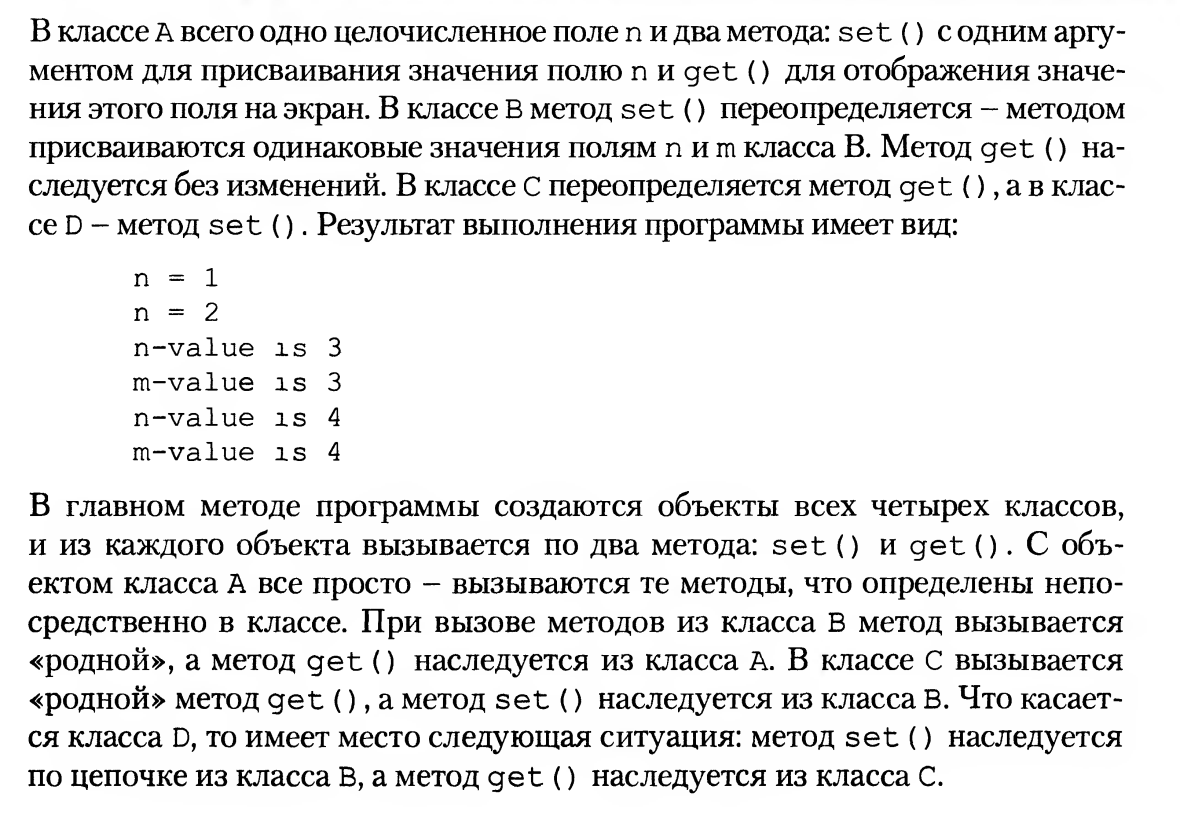
Рассмотрим пример многоуровнего наследования: класс А является базовым для класса В, а класс В, в свою очередь, является базовым для класса С, а класс С является базовым для класса D. При этом базовый класс А содержит виртуальные методы, которые переопределяются при наследовании.







В классе А всего одно целочисленное поле n и два метода: set () с одним аргументом для присваивания значения полю n и get () для отображения значения этого поля на экране. В классе В метод set () переопределяется – методом присваиваются одинаковые значения полям n и m класса В. Метод get () наследуется без изменений. В классе С переопределяется метод get (), а в классе D – метод set (). Результат выполнения программы:



В главном методе программы создаются объекты всех четырех классов, и из каждого объекта вызывается по два метода: set () и get (). С объктом класса А все просто – вызываются те методы, что определены непосредственно в классе.При вызове методов из класса В метод вызывается родной, а метод get () наследуется из класса А. В классе С вызывается родной метод get (), а метод set () наследуется из класса В. Что касается класса D. То имеет место следующая ситуация: метод se t() наследуется по цепочке из класса В, а метод get () наследуется из класса С.

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ № 7**

1. Что такое виртуальная функция (метод) при наследовании?
2. Чем отличается переопределение метода от его перегрузки при наследовании?
3. Что такое чисто виртуальная функция? Для чего она используется?
4. Синтаксис виртуальной функции?
5. Как называется класс, содержащий хотя бы одну чисто виртуальную функцию?
6. Приведите пример иерархической структуры при наследовании?
7. Как использовать виртуальные методы в многоуровневом наследовании?

**Лабораторная работа №8**

**Тема: Шаблоны**

**Задания:**

1. Создать шаблон функции, которая находит значение максимального элемента массива. Аргументами функции должны быть имя и размер массива (типа int). Создать шаблон функции, осуществляющей сортировку данных массива. Аргументами функции должны быть имя и размер массива (типа int). Продемонстрировать работу шаблонов на данных различных типов.

2. Создать шаблон класса, у которого есть 2 поля в виде одномерных массивов одного размера. Заполнить массивы с помощью генератора случайных чисел. Найти сумму элементов этих массивов. Продемонстрировать работу шаблонов на данных различных типов.

3. Создать шаблон класса «стек». Использовать его при решении задачи 3 из лабораторной № 1.

**Методические указания**

Для выполнения лабораторной работы № 7 требуется знать основные понятия: *шаблоны функции, перегрузка шаблона функции, явная специализация функции, шаблон класса, обобщенный класс,**перегрузка шаблона класса, явная специализация класса, типы по умолчанию.*

Часто при работе с данными разных типов используют один и тот же универсальный алгоритм. Для этого надо разработать единый программный код, реализующий общий алгоритм, а затем использовать его для каждой конкретной ситуации. Следует учесть, что в этой ситуации мы ранее использовали перегрузку функций. Для решения той же задачи и одновременно решения задачи сокращения листинга можно использовать шаблоны функций и классов. Шаблоны функций и классов – это наиболее мощный инструмент, без которого невозможно было бы написать серьезный код. Например, библиотека STL - это библиотека шаблонов.

**Шаблон функции (обобщенные функции) –** это функции, в которых тип данных передается, как формальный параметр, называется шаблоном функции.

В шаблоне функции конкретный тип, который используется в функции, заменен на название, например Т , вместо этого названия может быть подставлен любой тип. Синтаксис шаблона функции с аргументами одинакового типа:

Template <class **T**> **T** имя\_функции (**Т** аргумент1, **Т** аргумент2) {код функции}

Где: **Т** – это аргумент шаблона

Рассмотрим пример программы с шаблоном функции change() - обмен значениями аргументов функции.

*// шаблон функции*

template <class Т> void chang(Т &a, Т &b){

Т t; //ввели доп. переменную t обобщенного типа, как и у аргументов

t=a; //сохранили в ней значение первого аргумента

a=b; // 1-му аргументу присвоили значение второго аргумента

b=t; // 2-му аргументу присвоили значение введенной переменной t

}

int main() {

double x=2.3,y=4.4;

int m=5,n=6;

*//вызов шаблона функции*

chang(x,y); //вызов функции с аргументами типа double

cout<<"x: "<<x<<endl;

cout<<"y: "<<y<<endl;

chang(m,n); //вызов функции с аргументами типа int

cout<<"m: "<<m<<endl;

cout<<"n: "<<n<<endl;

return 0;

}

Обобщенная функция change () имеет два аргумента одного обобщенного типа, которые передаются по ссылке. В результате выполнения функции происходит обмен значениями аргументов функции (именно для этого аргументы передаются по ссылке). Это происходит для аргументов всех типов. В этой программе мы написали шаблон функции один раз и заставили его работать с разными типами данных. Схематичное изображение шаблона функции в памяти для этой программы:

template <class Т> void chang(Т &a, Т &b){

…….

}

void chang (double &a, double &b){

…….

}

void chang (int &a, int &b) {

…….

}

Один шаблон функции в листинге (исходном файле)

Тип аргументов определяет конкретную реализацию функции

**Две шаблонных функции в памяти**

**Две шаблонных функции в памяти**

Две шаблонных функции в памяти

Рис.4 Шаблон функции

Рассмотрим пример шаблона функции с двумя обобщенными типами данных.

template <class typel ,class type2>

void myfunc(typel x, type2 y){

cout<<x<<" "<<y<<endl;}

int main ()

{

myfunc(10,'g');myfunc(2.5,10L);

return 0;

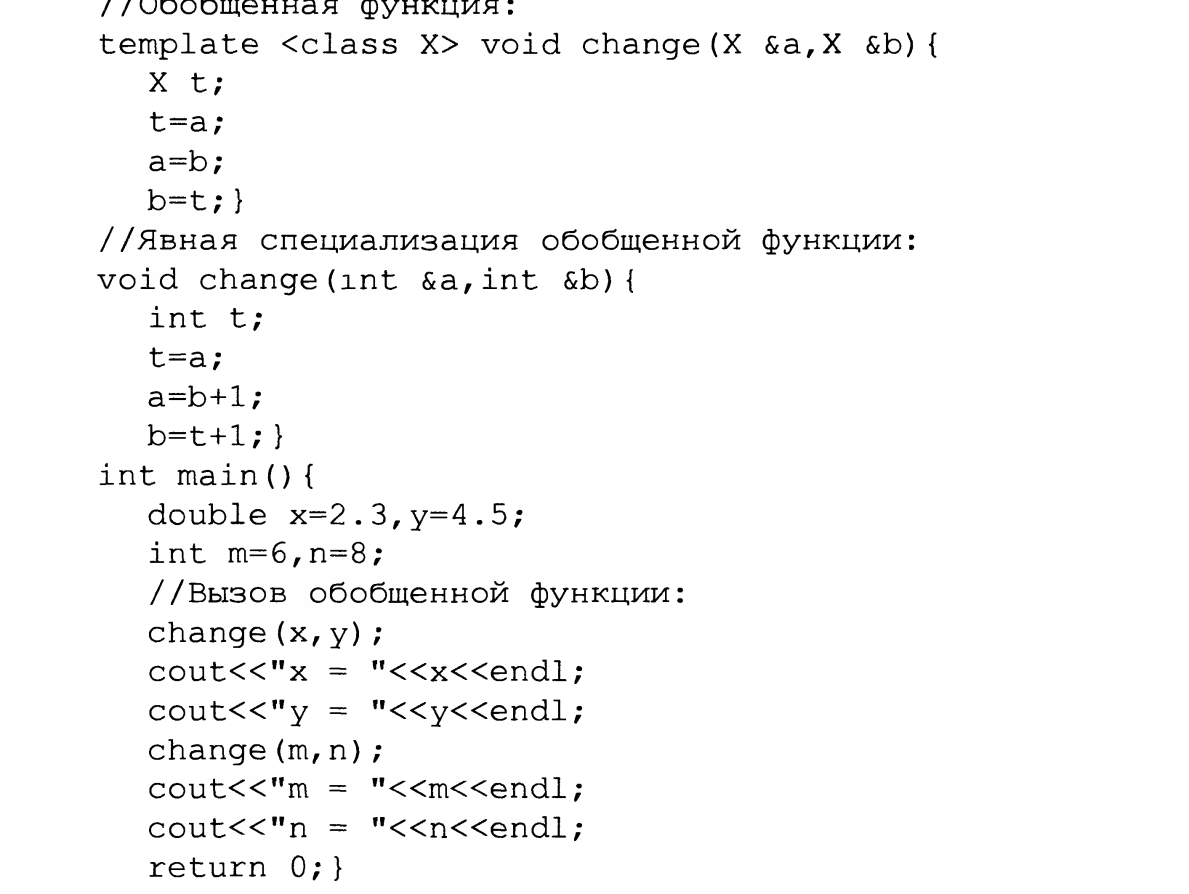
}

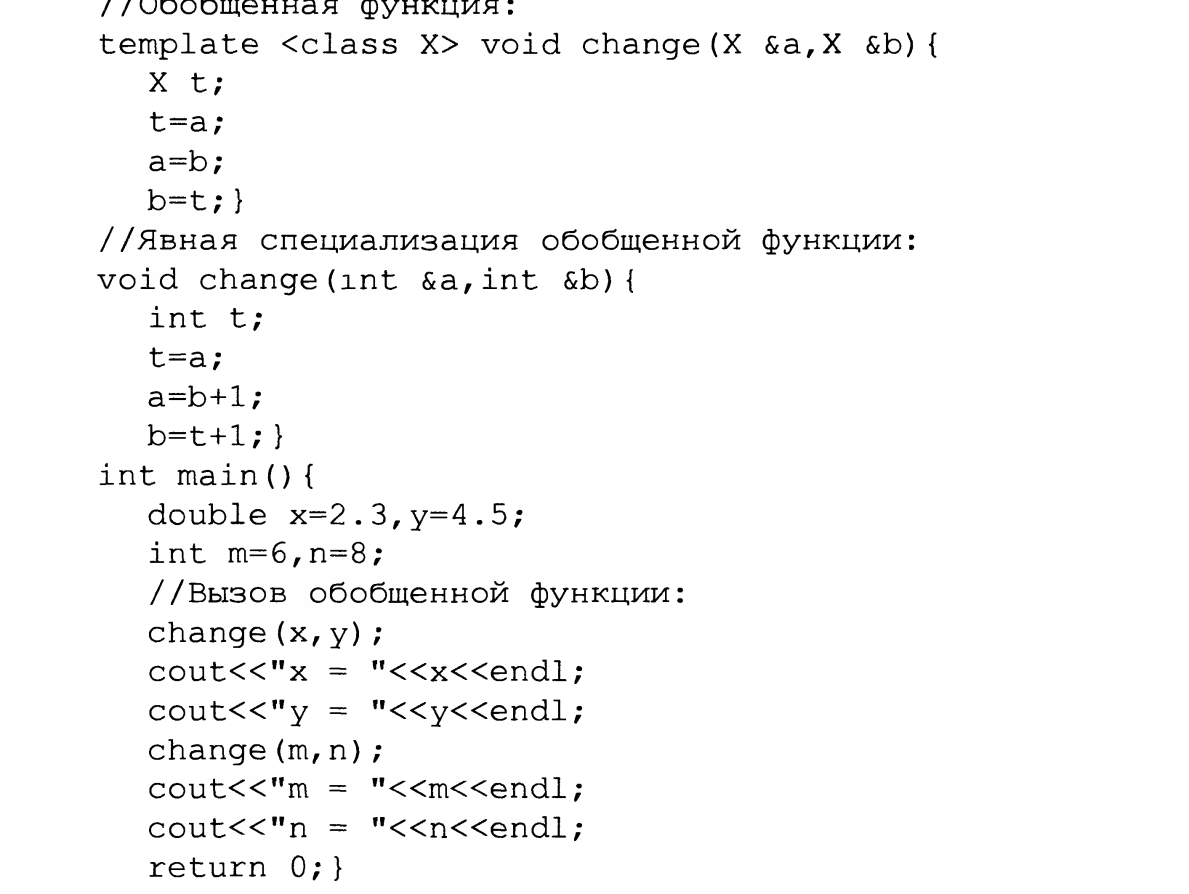
В данном примере при генерации конкретных экземпляров функции myfunc(), фиктивные имена типов typel и type2 заменяются компилятором на типы данных int и char\*или double и long соответственно.

**Перегрузка шаблона функции (явная специализация функции)**

Для обобщенных функций можно выполнять явную перегрузку, которая также называется явной специализацией. При явной перегрузке создается отдельный вариант этой функции с явным указанием типов аргументов и результата функции. Явная специализация функции – это фактически стандартное описание обычной, не обобщенной функции. Рассмотрим пример явной специализации обобщенной функции.

Перегрузим функцию chang():

****

****

В результате выполнения функции происходит обмен значениями аргументов функции (именно для этого аргументы передаются по ссылке). Это происходит для аргументов всех типов кроме типа int. На этот случай в программе создана специализация функции change (). Происходит не просто обмен значениями аргументов, а эти значения увеличиваются на единицу.

Таким образом, при вызове обобщенной функции реализуется один и тот же алгоритм во всех случаях, кроме ситуации, когда аргументы функции имеют тип int. В этом отношении явная специализация обобщенной функции позволяет создать исключение из правил – правилом в данном случае служит основной вариант обобщенной функции.

**Шаблон класса (обобщенный класс) – это** класс в которых тип данных передается как формальный параметр.

Синтаксис шаблона класса:

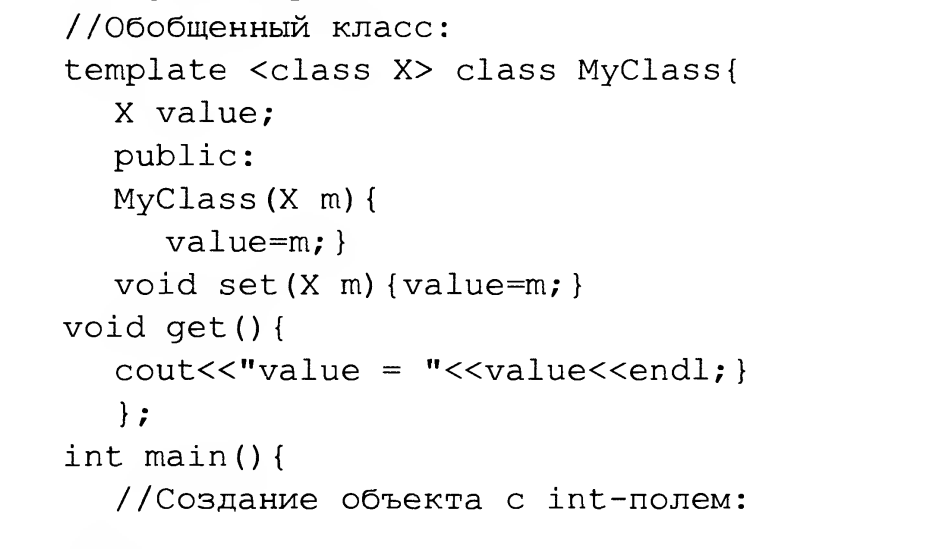
Template <class T>

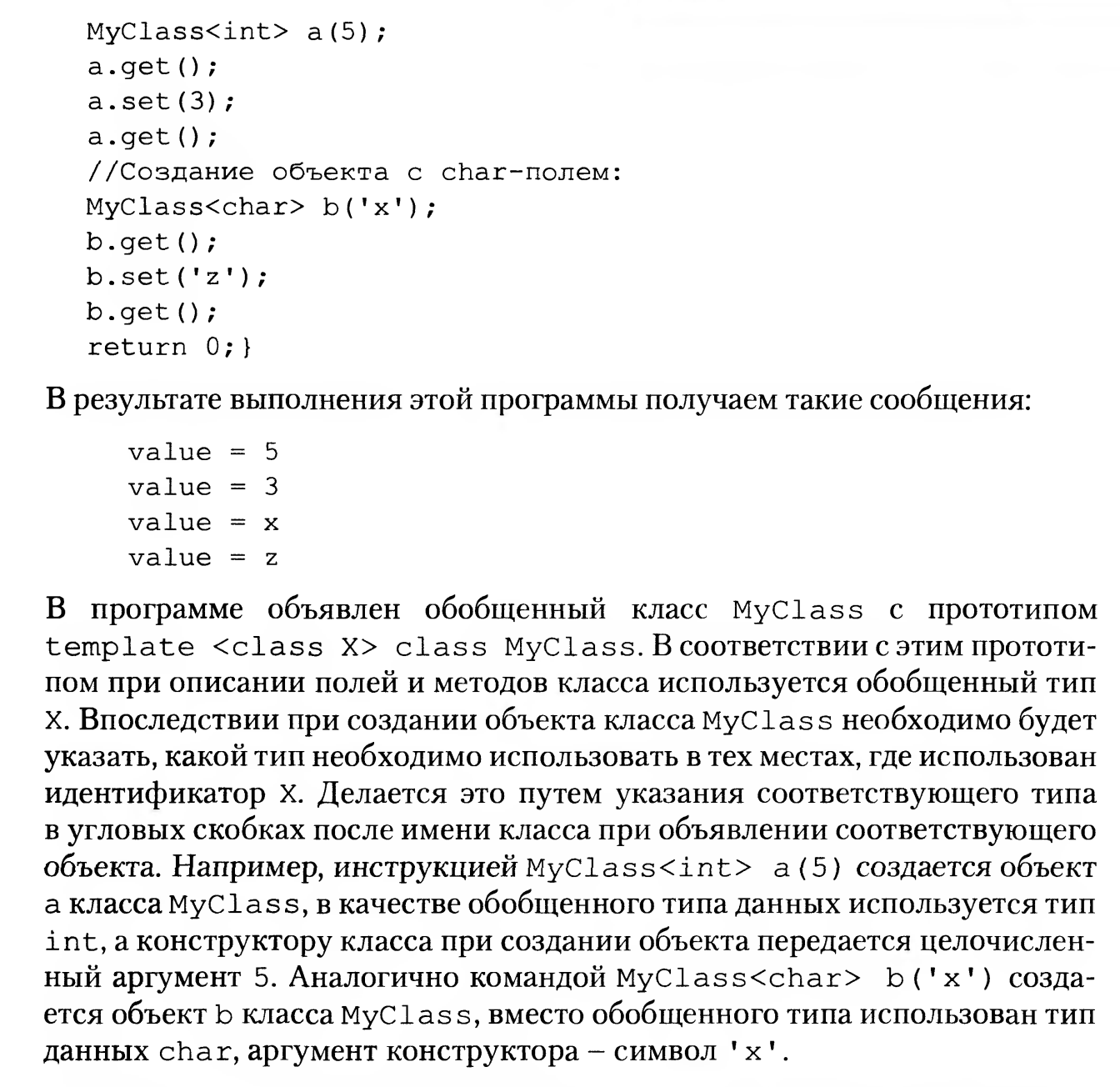
сlass имя\_класса{

//Данные и методы используют аргумент шаблона Т

};

Рассмотрим программу с шаблоном класса:





У обобщенного класса MyClass закрытое поле value типа x, конструктор с одним аргументом того же типа и два метода: set () с аргументом типа Х для изменения значения закрытого поля класса и метод get () без аргументов для отображения значения закрытого поля.

В программе объявлен обобщенный класс MyClass с прототипом template <class X> class MyClass. В соответствии с этим прототипом при описании полей и методов класса используется обобщенный тип Х.

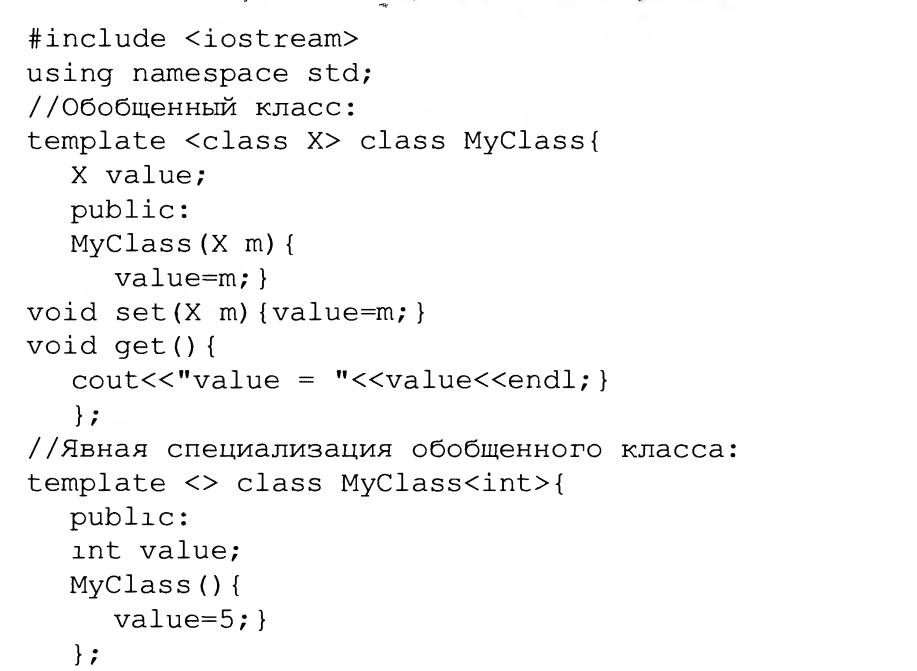
При создании объекта класса указываем в угловых скобках после имени класса, какой тип использовать в качестве обобщенного. Например, инструкцией MyClass <int> a(5) создается объект **а** класса MyClass, в качестве обобщенного типа данных используется тип int, а конструктору класса при создании объекта передается целочисленный аргумент 5. Аналогично, командой <char> (‘x’) создается объект b класса MyClass, вместо обобщенного типа использован тип данных char, аргумент конструктора – символ ‘х’.

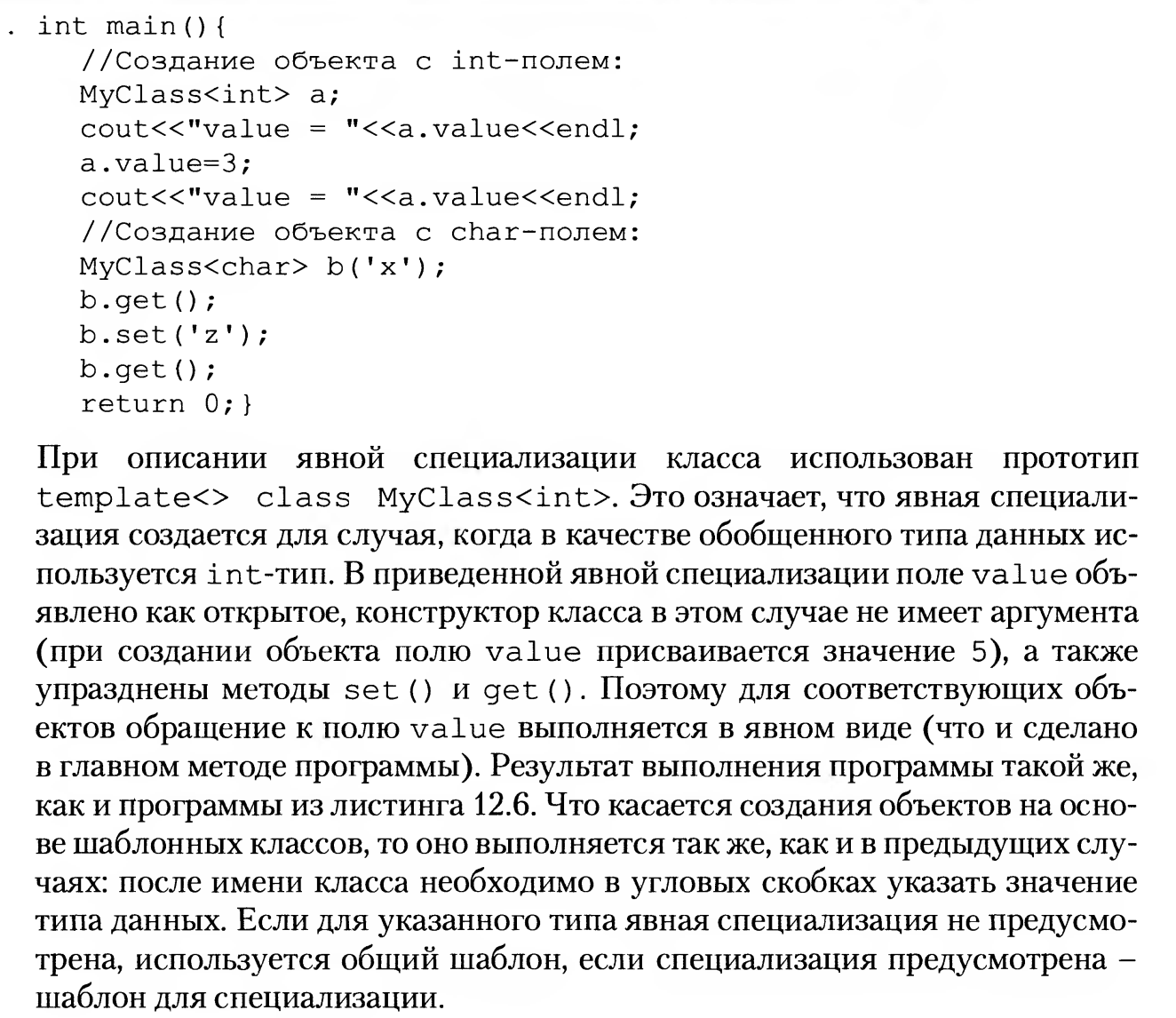
В этой программе мы написали шаблон класса один раз и заставили его работать с разными типами данных и методов.

**Перегрузка шаблона класса (явная специализация класса)**

Как и для обобщенных функций, для обобщенных классов можно создавать явные специализации. Явная специализация позволяет создать исключение из общего правила, определяемого шаблоном обобщенного класса.

При создании явной специализации обобщенного класса после ключевого слова template, с которого начинается описание специализации, следуют пустые угловые скобки. Далее следует ключевое слово class, название класса и в угловых скобках – тип (или типы) данных, для которых выполняется явной специализации класса. Рассмотрим пример создания явной специализации класса.





**Типы по умолчанию**

С одной стороны, при создании объекта на основе обобщенного класса нет возможности автоматически определять значения для обобщенных типов, как это происходит в обобщенных функциях (на основе аргументов). Поэтому значения для обобщенных типов следует указывать в явном виде. С другой стороны, такой подход не очень удобен, поскольку приводит к усложнению процессу объявления объектов. В некоторых случаях разумно использовать значения для обобщенных типов, используемые по умолчанию. Рассмотри листинг программы.

Речь идет о следующей ситуации. Создается обобщенный класс, в котором для обобщения типов предусмотрены значения по умолчанию. Если затем при создании объекта обобщенный тип данных явно не указан (хотя угловые скобки все равно придется указывать), то используется значение типа по умолчанию. Для того чтобы задать обобщенный тип данных по умолчанию, необходимо в прототипе класса в том месте, где вводится индикатор обобщенного типа, указать через знак равенства тип по умолчанию. Рассмотрим листинг программы.

template <class X=int> class MyClass {

X value;

public:

MyClass(X m){

value=m;}

void set(X m) {value=m;}

void get(){

cout<<"value= " <<value<<endl;

system("pause");}

};

int main()

{

//создание объекта с int - полем

MyClass<> a(5);

a.get();

a.set(3);

a.get();

//Создание объекта с char-полем

MyClass<char> b('x');

b.get();

b.set('z');

b.get();

return 0;}

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №8**

1. Что такое обобщенная функция?

2. Шаблоны функции позволяют удобным способом создать семейство

а) переменных; б) функций; в) классов; г) программ.

3. Истинно ли утверждение, что шаблонная функция может иметь несколько аргументов?

4. Реальный код шаблонной функции генерируется при:

а) объявлении функции в исходном коде;

б) создании объекта;

в) запуске функции из её исходного кода.

1. Что такое обобщенный (шаблонный) класс?
2. В каком случае его имеет смысл применять?
3. Шаблонный класс:

а) работает с разными типами данных;

б) генерирует идентичные объекты;

в) генерирует классы с различным числом методов

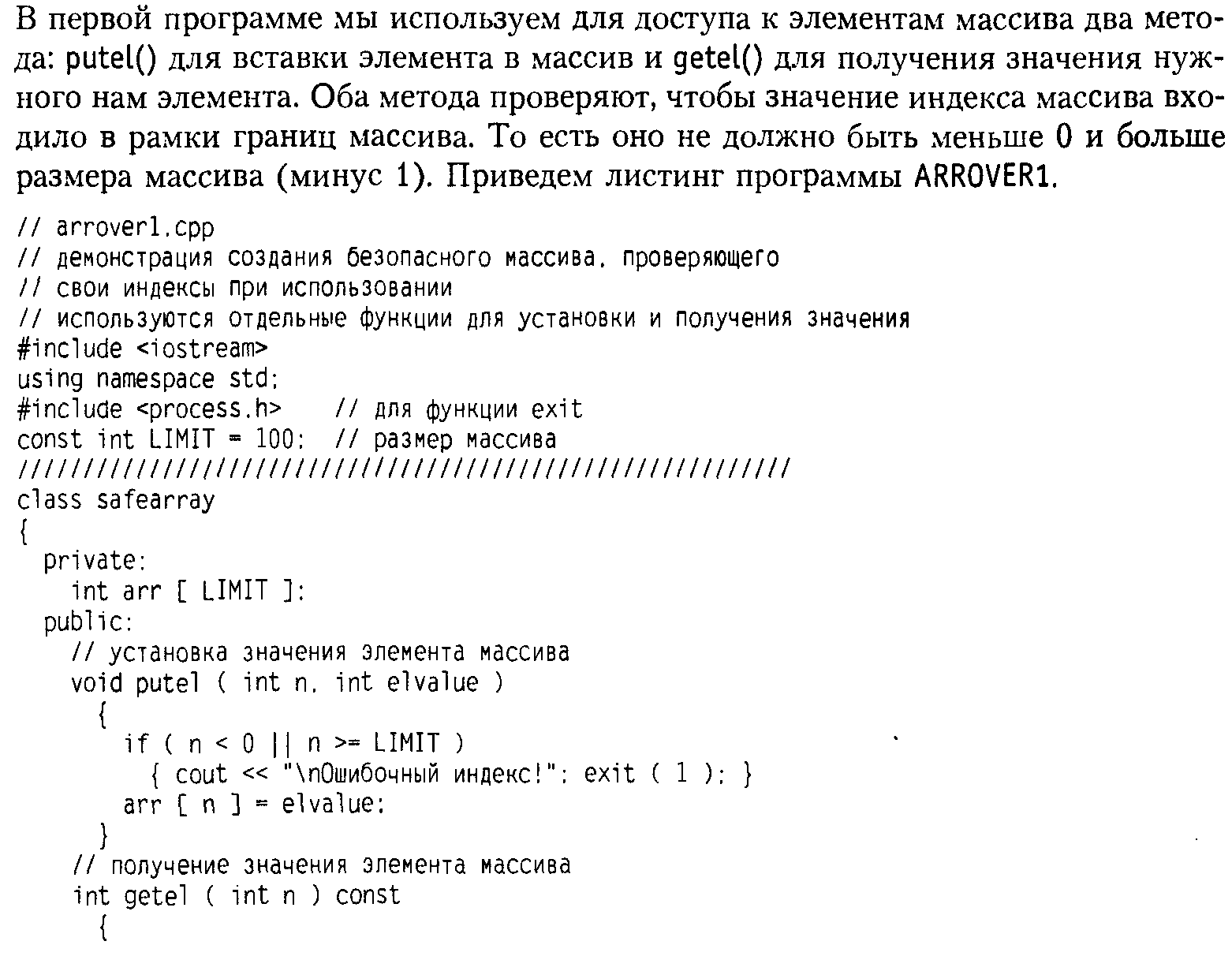
**Лабораторная работа № 9**

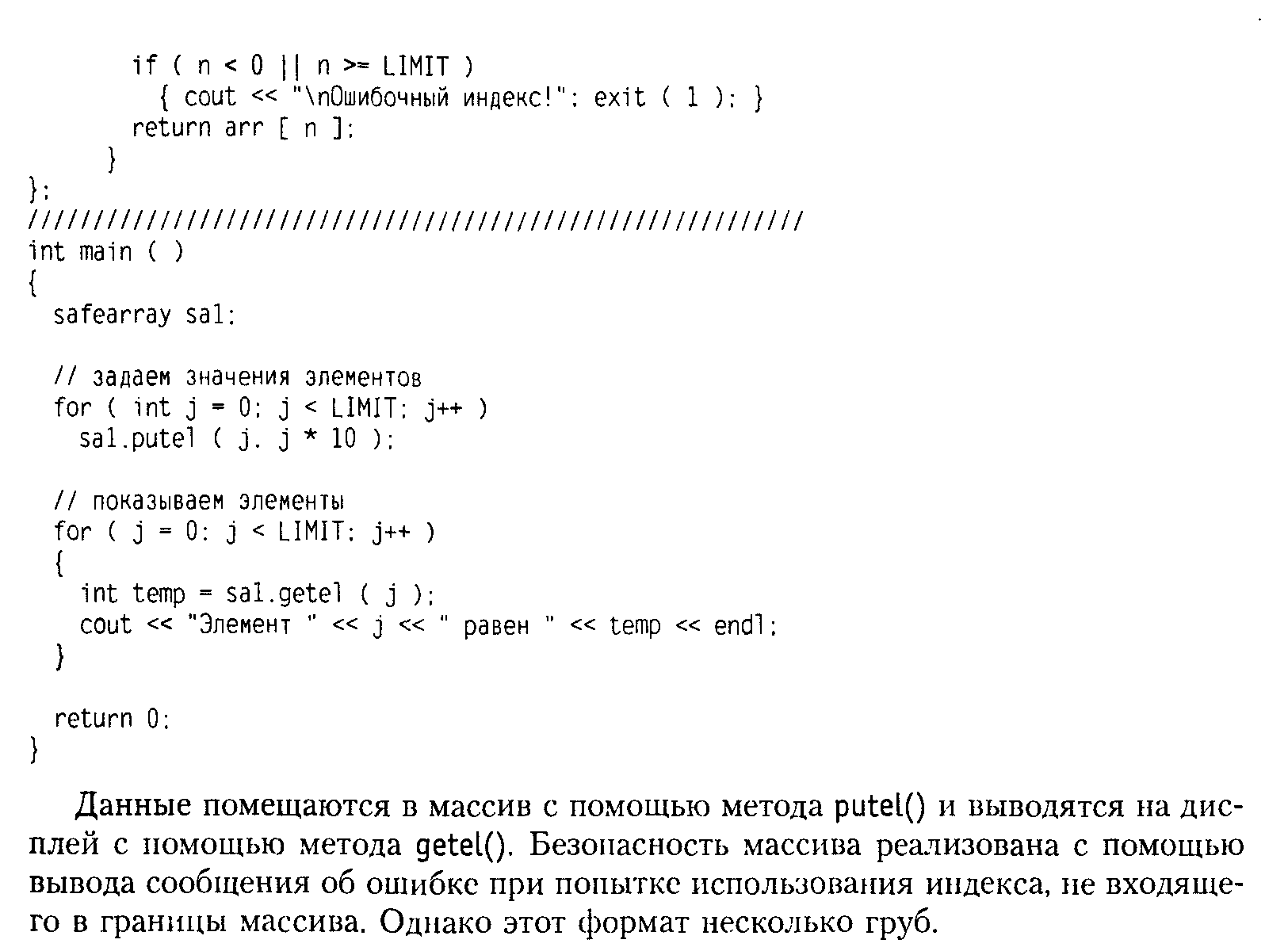
**Тема: Исключения**

**Задания:**

**1.**Добавьте класс исключений к программе ARROVER1, чтобы индексы, выходящие за пределы массива, вызывали генерацию исключения. Блок-ловушка может выводить пользователю сообщение об ошибке.

Листинг программы arrover1:





Данные помещаются в массив с помощью метода putel () и выводятся на экран с помощью метода getel (). Безопасность массива реализована с помощью вывода сообщений об ошибке при попытке использования индекса, не входящего в границы массива.

1. Задание 3 из лабораторной работы №4 выполнить с использованием механизма исключений.

**Методические указания**

Для выполнения лабораторной работы № 9 требуется знать основные понятия: *исключения, механизм исключений, генерация исключения, блок повторных попыток, обработчик ошибок.*

**Исключения –** это концепция защиты программы от ошибок. Концепция программирования с защитой от ошибок предполагает включение в код программного модуля средств контроля его нормального выполнения. Это обеспечивает живучесть и корректную работу программ. Ошибки, возникающие во время работы программы, могут быть вызваны выходом за пределы памяти, ошибкой открытия файла, инициализацией объекта недопустимым значением или использованием индекса, выходящего за пределы вектора, деление на ноль, указание несуществующего адреса памяти.

**Синтаксис исключений**

Представим себе приложение, создающее объекты какого-либо класса и работающее с ними. В нормальной ситуации вызовы методов не приводят к ошибкам. Но иногда в программе возникает ошибка, которую обнаруживает метод. Он информирует программу об ошибке. При использовании исключений это событие называется *генерацией исключительной ситуации*. В приложении имеется отдельная секция кода, в которой находится операции по обработке ошибок. Этот код называется *обработчиком исключительных ситуаций* или *улавливающим блоком*. В нем отлавливаются исключения, сгенерированные методами. Любой код приложения, использующий объекты класса, заключается в *блок повторных попыток*. Соответственно, ошибки, возникшие в последнем, будут пойманы улавливающим блоком. Код, который не имеет отношения к классу, не включается в блок повторных попыток. Рассмотрим схему механизма исключений.

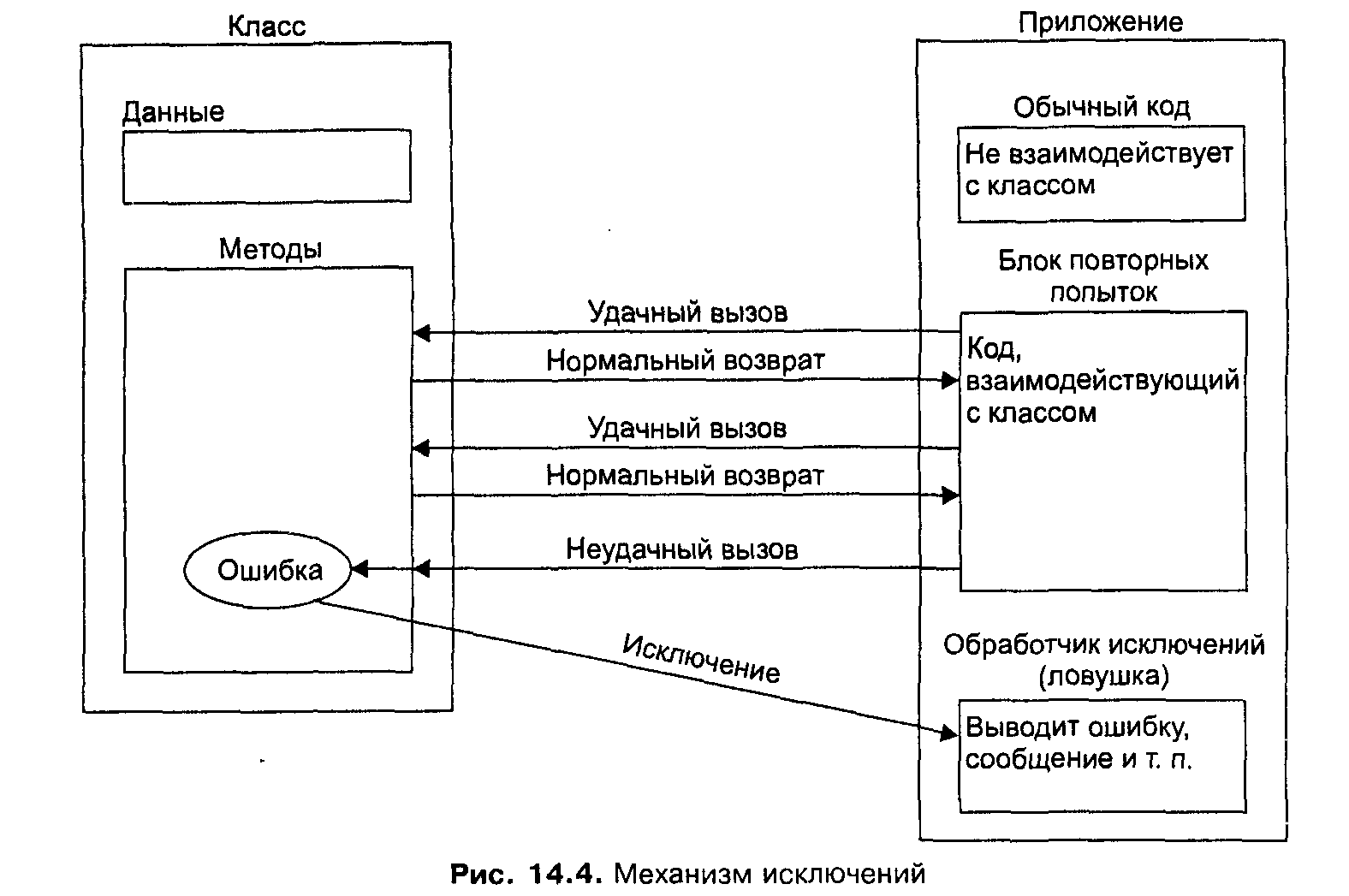
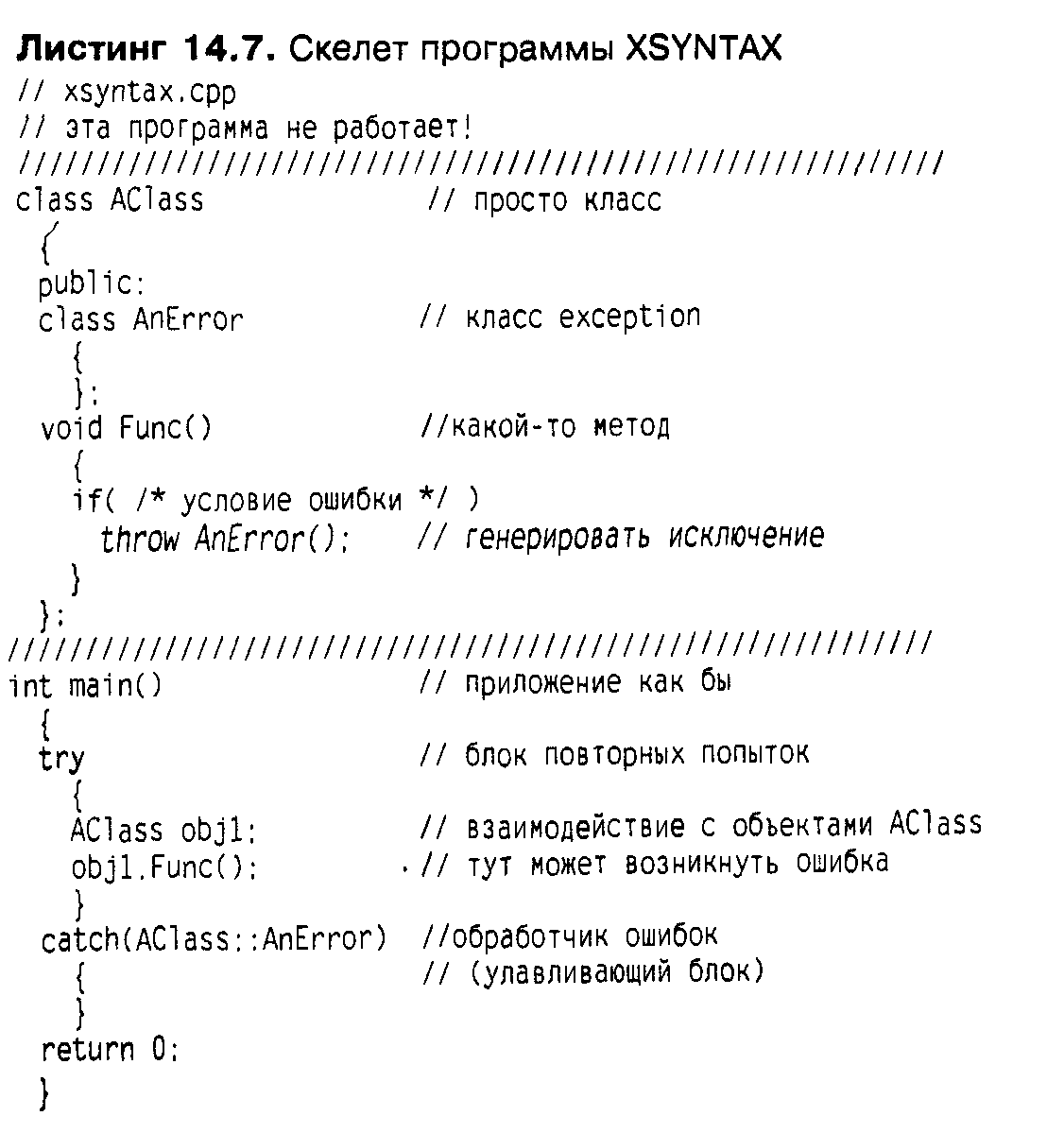


Рис. 3 Механизм исключений

Механизм исключений использует три ключевых слова: **catch, throw, try**. Для наглядности синтаксис исключений покажем с помощью скелета программы:



Мы начинаем работу с создания класса AClass, представляющего собой класс, в котором могут возникнуть ошибки. Класс исключений AnError определен в общедоступной части класса AClass. Методы последнего подвергаются тщательному обследованию на предмет выявления ошибок. Если таковые обнаруживаются, мы с помощью ключевого слова trow и конструктора класса исключений производим генерацию исключительной ситуации:

trow AnError();

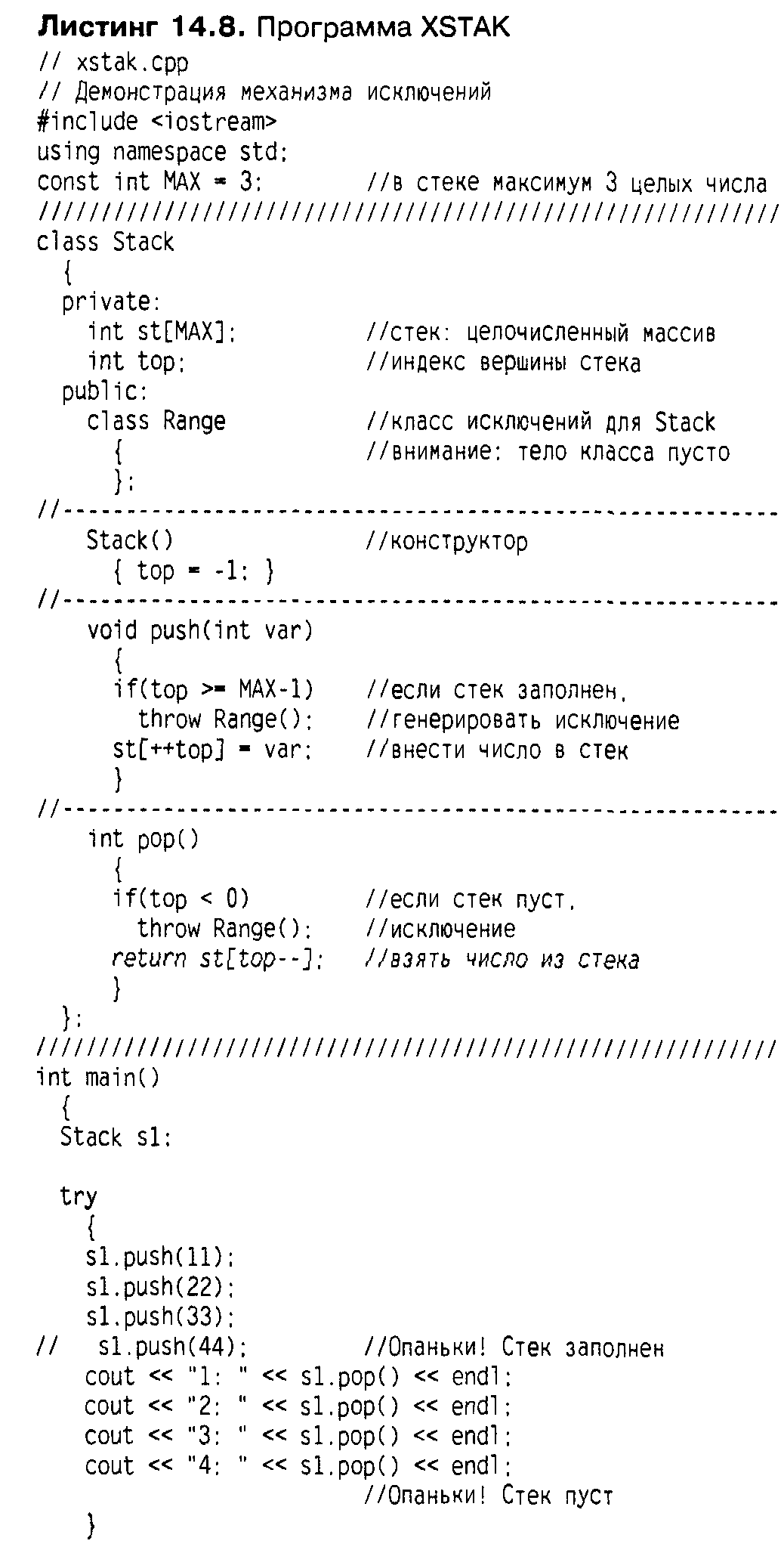
В main() мы заключаем все выражения, каким-либо образом относящиеся к AClass, в блок повторных попыток. Если таковые из них приводят к тому, что возникают подозрения на наличие ошибки в методе AClass, то генерируется исключение, и управление программой переходит к улавливающему блоку, который следует сразу же за блоком повторных ошибок

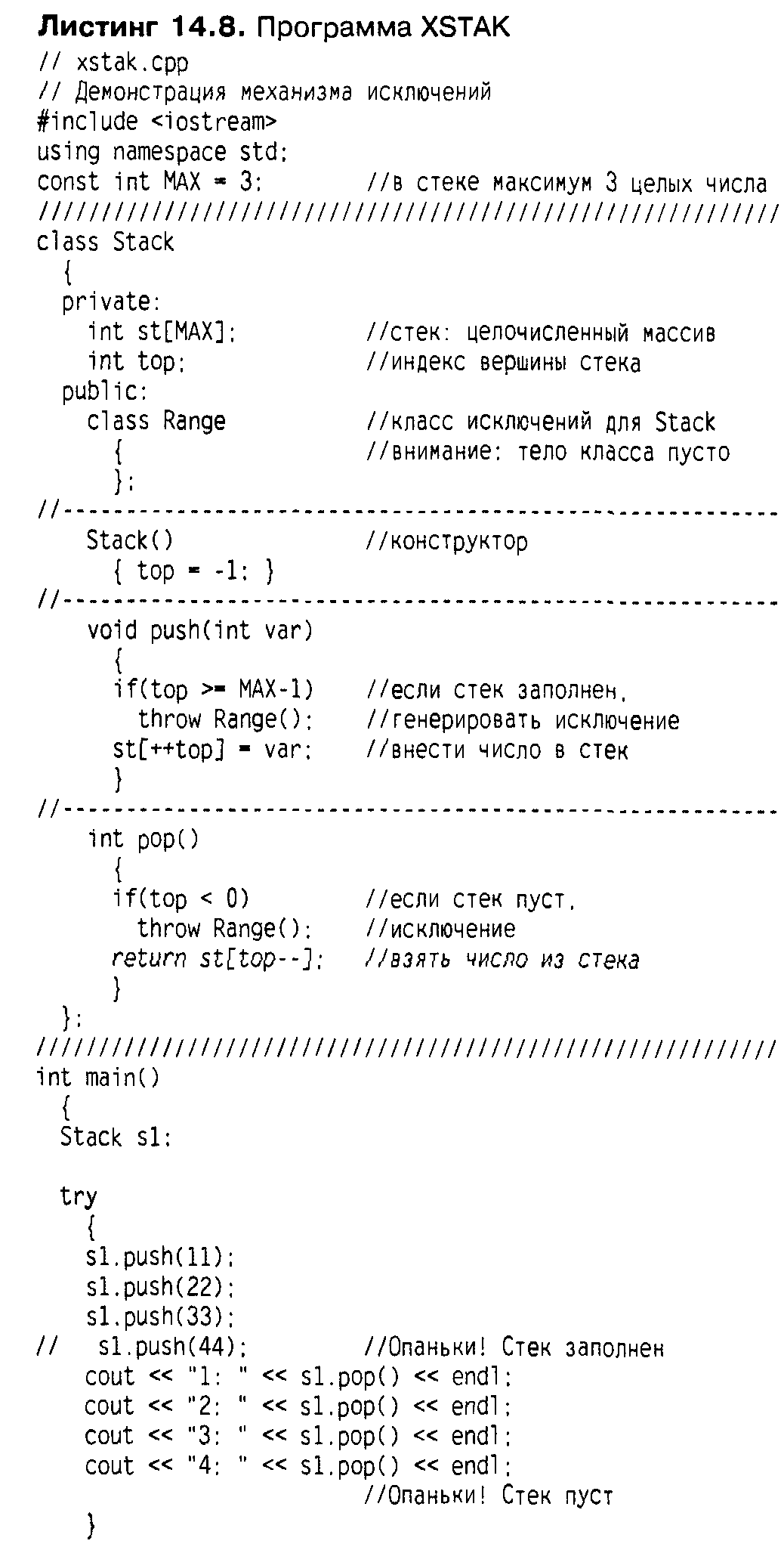
**Последовательность событий**

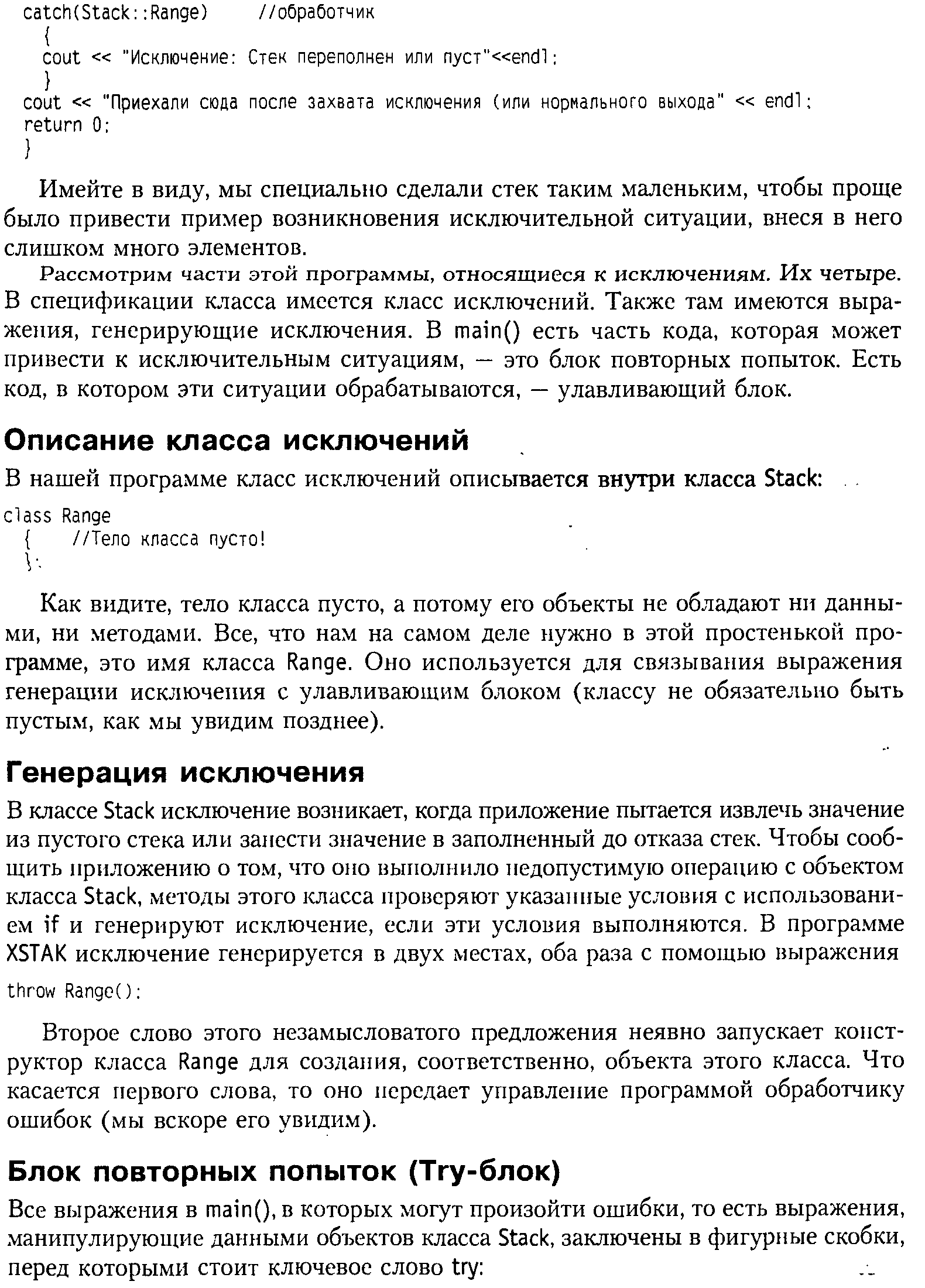
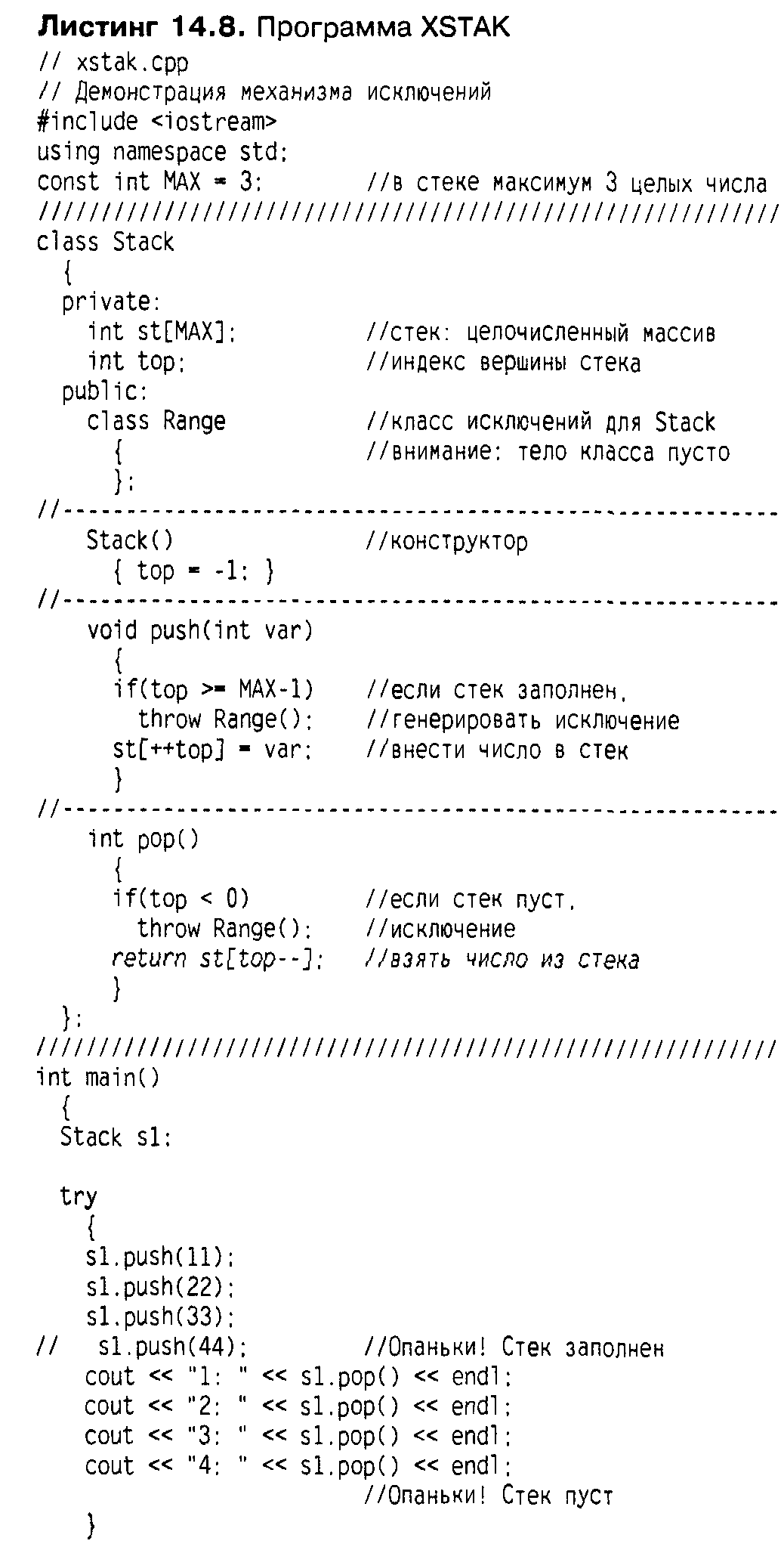
Давайте подведем итоги и покажем последовательность действий программы при возникновении ошибок:

1. Код нормально выполняется вне блока повторных попыток.
2. Управление переходит в блок повторных попыток.
3. Какое-то выражение в этом блоке приводит к возникновению ошибок в методе.
4. Метод генерирует исключение.
5. Управление переходит к обработчику ошибок (улавливающему блоку), следующему сразу за блоком повторных попыток.

Рассмотрим листинг программы создания стека в виде целочисленного массива. Использовать исключения для обнаружения ошибок при помещении в стек или извлечении из стека слишком большого количества элементов.







Рассмотрим части программы, относящиеся к исключениям.

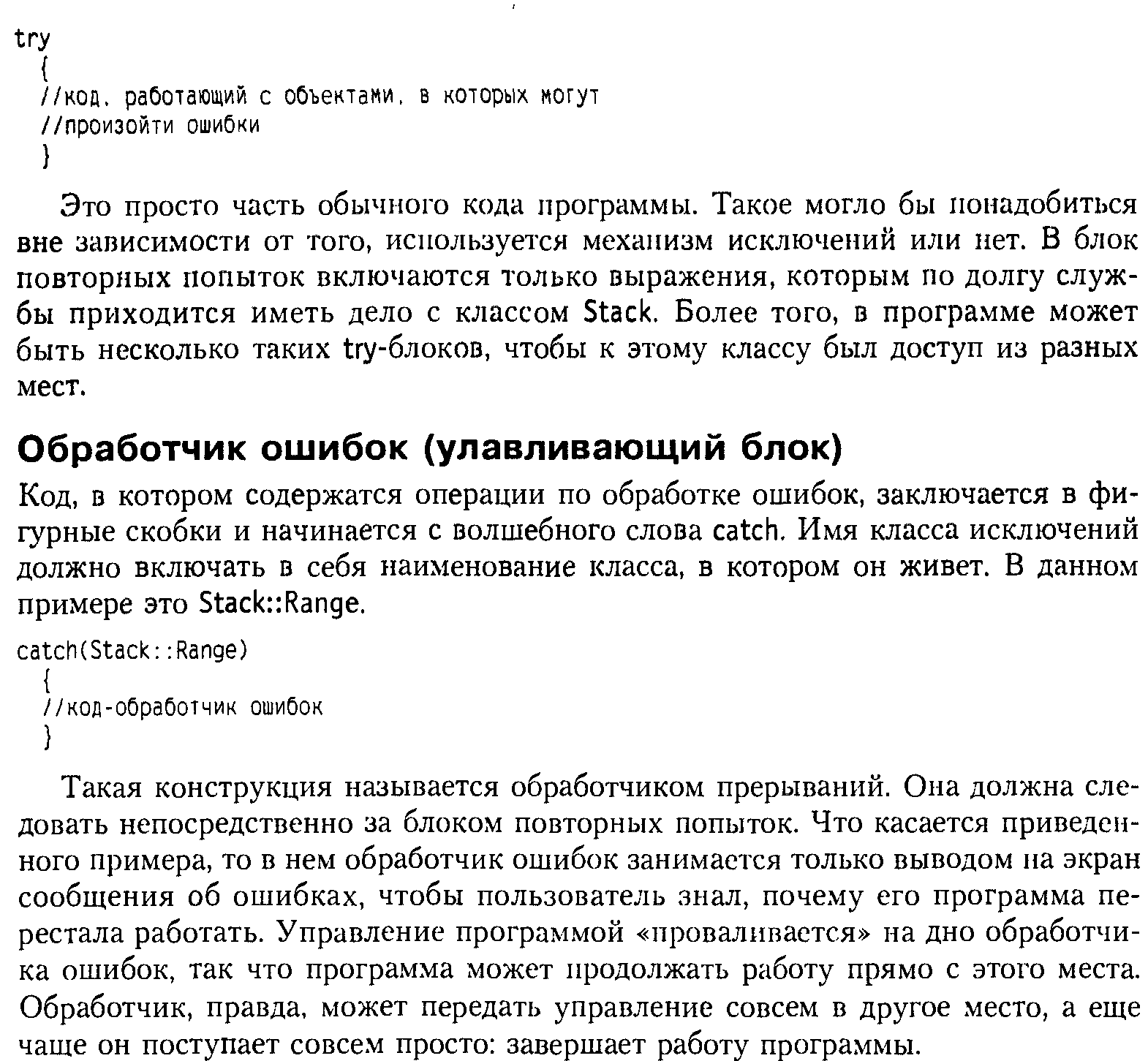
**Генерация исключения**

В классе исключение возникает, когда приложение пытается извлечь значение из пустого стека или занести значение в заполненный до отказа стек. Чтобы сообщить приложению о том, что оно выполнило недопустимую операцию с объектом класса Stack, методы этого класса проверяют указанные условия с использованием if и генерируют исключение, если эти условия выполняются. В программе xstak исключение генерируется в двух местах, оба раза с помощью выражения

throw Range();

**Блок повторных попыток (Try-блок)**

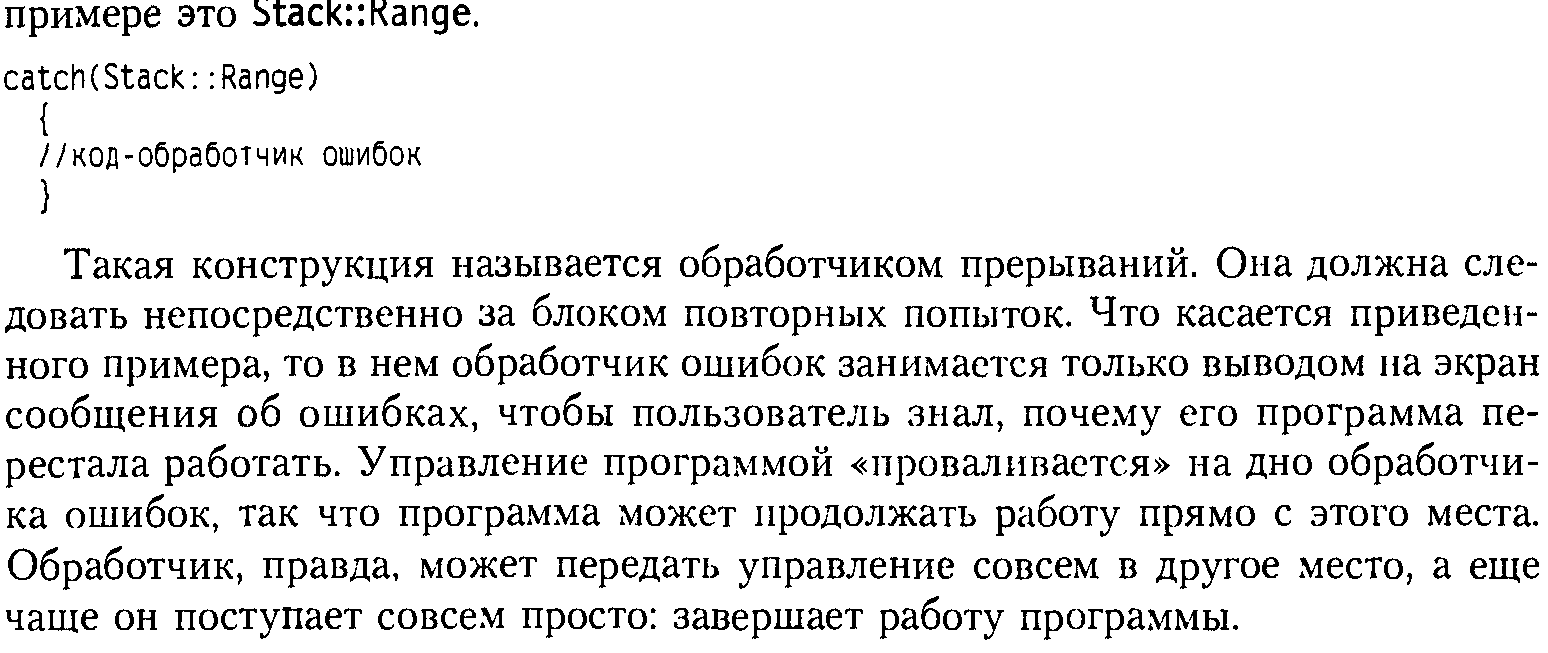
Все выражения в main (), в которых могут произойти ошибки, то есть выражения, манипулирующие данными объектов класса Stack, заключены в фигурные скобки, перед которым стоит ключевое слово **try:**



Это просто часть обычного кода программы. В блок повторных попыток включаются только выражения, которые имеют дело с классом Stack. В программе может быть несколько try-блоков, чтобы к этому классу был доступ из разных мест.

**Обработчик ошибок (улавливающий блок)**

Код, в котором содержаться операции по обработке ошибок, заключаются в фигурные скобки м начинается со слова catch. Имя класса исключений должно включать в себя наименование класса, в котором он живет. В данном примере это Stack::Rang:



Такая конструкция называется обработчиком прерываний. Она должна следовать сразу за блоком повторных попыток. В приведенном листинге обработчик ошибок занимается только выводом на экран сообщения об ошибках, чтобы пользователь знал, почему его программа перестала работать.

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №9**

1. Исключение в большинстве случаев возникает из-за:

* Программиста, написавшего исходный код приложения;
* Создателя класса, написавшего его метод;
* Ошибка выполнения;
* Сбоя в операционной системе.

1. При работе с механизмом исключений в С++ используются следующие ключевые слова: \_\_\_\_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.
2. Исключения передаются:

* Из блока-ловушки в блок повторных попыток;
* Из выражения, создавшего исключительную ситуацию, в блок повторных попыток;
* Из точки, где возникла ошибка, в блок-ловушку.
* Из выражения, в котором возникла ошибка, в блок-ловушку.

1. Для следующих ошибок обычно генерируются исключения:

* Чрезмерное количество данных грозит переполнить массив;
* Пользователь нажал Ctr+C для закрытия программы;
* Скачок напряжения в сети привел к перезагрузке системы;
* «New» не может зарезервировать необходимый объем памяти.

1. Истинно ли утверждение о том, что программа может продолжить свое выполнение после возникновения исключительной ситуации?
2. Поясните механизм исключений.

**Лабораторная №10**

**Тема: Статические поля и методы класса**

**Задание:**

Написать программу для решения статическими методами следующей задачи. Из зенитного орудия производится стрельба по самолету. Вероятность попадания одного выстрела равна p. Стрельба производится n раз или до первого попадания. В программе необходимо создать класс с полем p и методом (n – аргумент метода) для вычисления результата стрельбы (0 если самолет не сбит, 1 если цель уничтожена). Предусмотреть метод для вычисления оценочного значения вероятности поражения цели (в пределе это значение стремиться к величине 1-(1-p)n)).

**Методические рекомендации**

Для выполнения лабораторной работы № 9 требуется знать основные понятия: *статическое поле, автоматическое поле, статические методы класса.*

**Статическое поле класса -** это поле, которое используется для хранения данных, совместно используемых объектами класса.

Например, пусть мы моделируем автомобильную гонку, и объекты — это гоночные автомобили. Нужно, чтобы каждый гонщик знал, сколько автомобилей участвует в гонке. В этом случае создаем статическую переменную класса, значение которой определяется количеством автомобилей в гонке. Эта переменная будет доступна всем объектам класса, и все они будут видеть одно и то же значение.

Статическое поле по своим характеристикам сходно со статической переменной: оно видимо только внутри класса. В отличие от обычного поля класса, статическое поле не исчезает при удалении объекта. Если поле данного класса описано с ключевым словом *static*, то значение этого поля будет одинаковым для всех объектов данного класса.

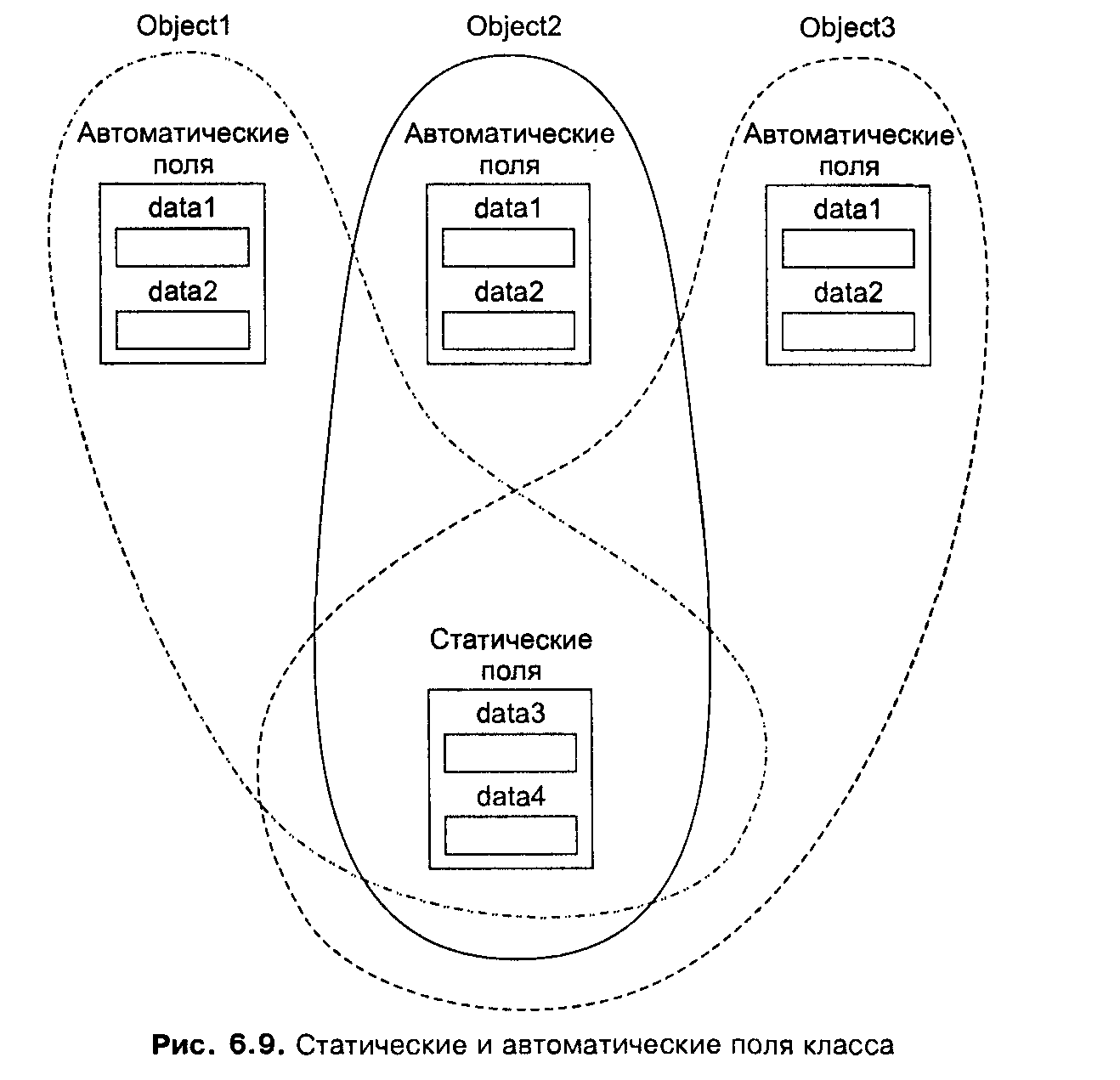
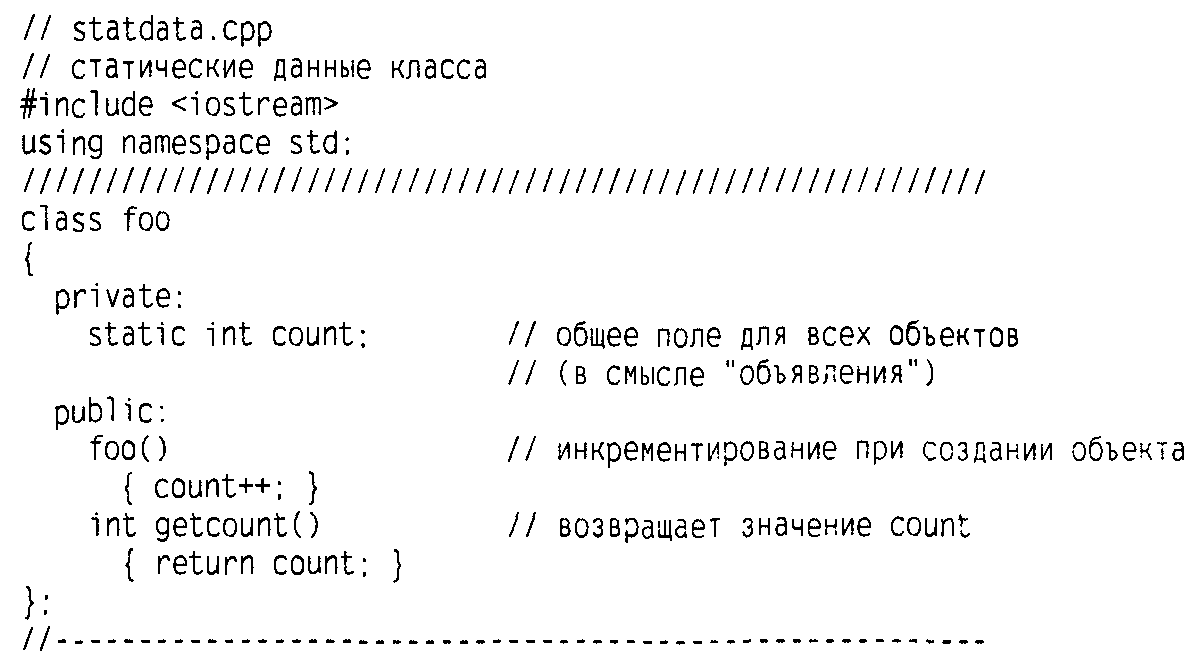
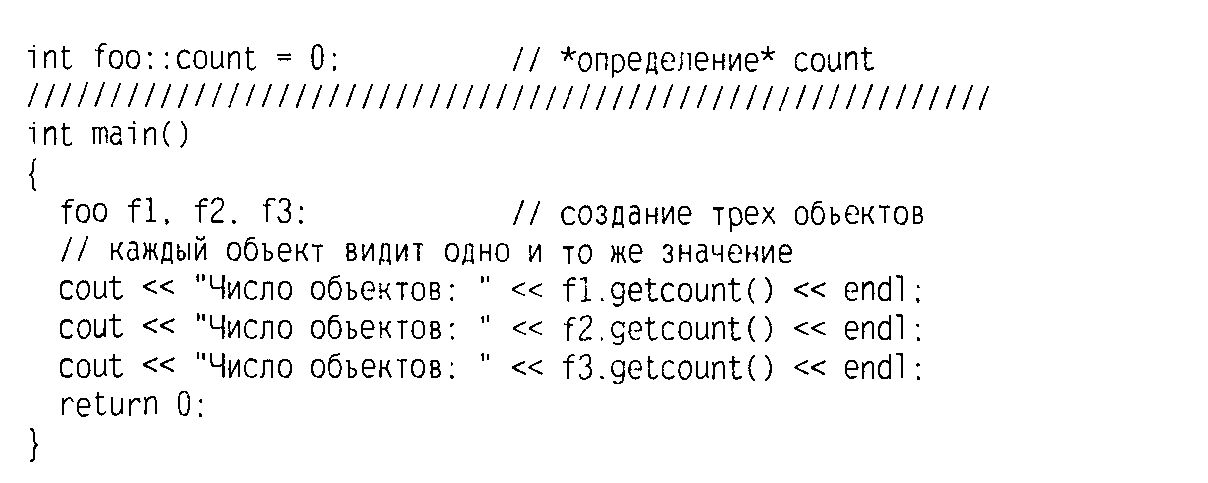


Рис.5 Статические и автоматические поля класса

Пример

****

****

В этом примере **foo** содержит единственное поле **count**, имеющее тип **static int.** Конструктор класса инкрементирует значение поля **count.** В функции main () мы определяем три объекта класса foo; Поскольку конструктор здесь вызывается трижды, инкрементирование поля **count** также происходит трижды. Метод getcount () возвращает значение count. Мы вызываем этот метод для каждого из объектов. И во всех случаях он возвращает одну и ту же величину:

*Число объектов: 3*

*Число объектов: 3*

*Число объектов: 3*

Если бы мы использовали не статическое, а автоматическое поле count, то конструктор увеличивал бы на единицу значение этого поля для каждого объекта, и результат работы программы выглядел бы следующим образом:

*Число объектов: 1*

*Число объектов: 1*

*Число объектов: 1*

**Раздельное определение статических полей класса**

Определение статических полей происходит не так, как для обычных полей. Обычные поля объявляются (компилятору сообщается имя и тип поля) и определяются (компилятор выделяет память для хранения поля) при помощи одного оператора. Для статических полей эти два действия выполняются двумя разными операторами: объявление поля находится внутри определения класса, а определение поля располагается вне класса, и зачастую представляет собой определение глобальной переменной.

В примере статическая переменная объявляется в классе, как поле *static int count***.** Определение поля static должно быть записано вне класса в глобальной области *int foo::count=0,* чтобы память под переменную выделялась один раз. Объекты уже не будут иметь свой экземпляр поля **count.** Т.е. статическое поле общее для всех объектов, как глобальная переменная.

**Статические методы класса**

Статические методы предназначены для обращения к статическим полям класса. Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса. Обращение к статическим методам производится так же, как к статическим полям: либо через имя класса, либо через имя объекта, если он уже создан.

Рассмотри листинг:

class A{

static int count; // поле скрытое

public:

static void inc\_count () { count ++ }

};

int A::count; // Определение поля в глобальной области

void main (){

A a;

// a. count++ - нельзя, поле count- скрытое

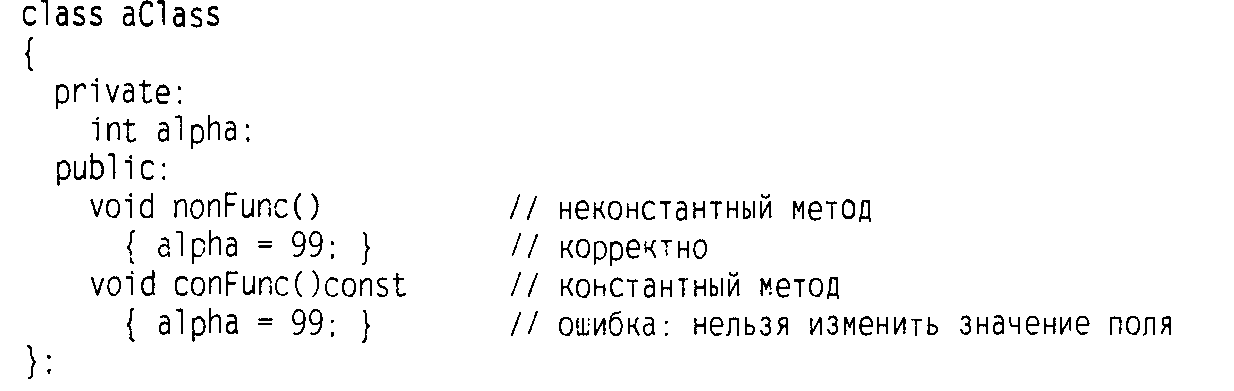
//Изменение поле с помощью статического метода

1. inc\_count (); // или A:: inc\_count ();}

**Константные методы**

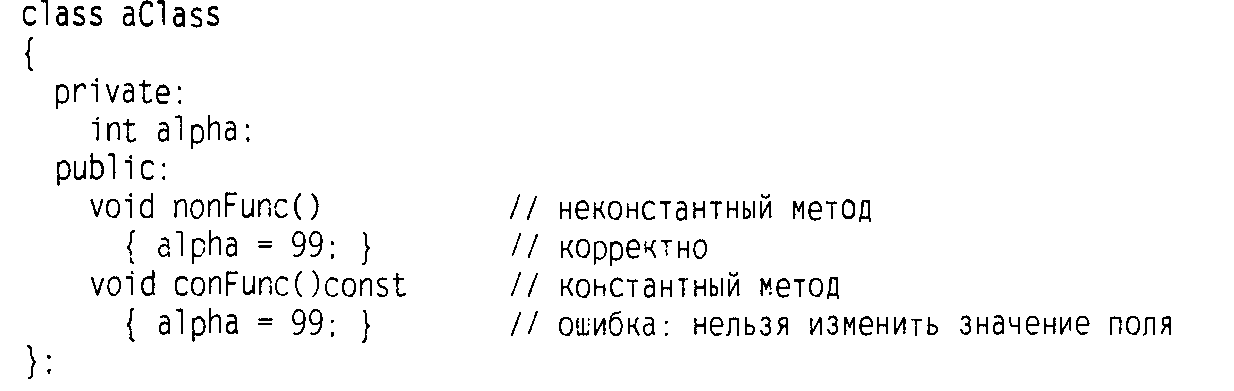
Статические методы не могут быть константными или виртуальными.

В **константных** методах можно использовать ключевое слово **const** для защиты от изменения значений переменных стандартных типов, таких как, например, **int**.



Обычный метод *nonFunc ()* может изменить значение поля alpha, а константный метод conFunc() не может. Для того чтобы сделать функцию константной, необходимо указать ключевое слово const после прототипа функции, но до начала тела функции.

Если объявление и определение функции разделены, то модификатор const нужно указать дважды, как при определении фенкции, так и при её объявлении. Те методы, которые только считывают данные из поля класса, имеет смысл делать константными, поскольку у них нет необходимости изменять значения полей объектов класса. Например:



};

aClass void conFunc () const{

return alpha

}

Int main(){

…..

}

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ № 10**

1. Что такое статическое поле класса? Как оно используется объектами класса?
2. В чем отличие статического поля от статической переменной?
3. В чем отличие статического поля класса от автоматического (обычного) поля?
4. Чем отличается определение статического и обычного полей?
5. Для чего используют статические методы класса?
6. Что такое константные методы? Для чего их используют?

**Литература**

1. Абрамов С.А., Гнездилова Г.Г., Капустин Е.Н., Селюн М.И. Задачи по программированию. – М.: Наука, 1988.
2. Абрамов С.А. Решебник задач по программированию, интернет сайт: <http://www.twirpx.com/file/756405/>
3. Р. Лафоре. Объектно-ориентированное программирование в С++. - СПБ.: Питер, 2004.
4. Дейтел Х.М., Дейтел П. Дж. Как программировать на С++. М.: Бином, 2000.
5. Павловская Т.А., Щупак Ю.А. Программирование на языке высокого уровня С/С++. –М.: Питер, 2003.
6. Либерти Дж. Освой самостоятельно С++ за 21 день. – М.: «Вильямс», 2000.
7. Кнут Д., Искусство программирования для ЭВМ. Том 3, 1975.
8. Шилдт Г., Теория и практика С++. Санкт-Петербург, 1999.
9. Стивен Прата, Язык программирования С++. Лекции и упражнения. Учебник. Изд. Dia Soft, Москва-Питер-Киев, 2005.