Содержание

1 Основные модели ООП

1.1 Основные идеи ООП

1) Основными сущностями программы являются **объекты**, всеми своими свойствами соответствующие сущностям решаемой задачи.
2) Программа есть описание **обмена сообщениями** между объектами.

1.2 Основные подмодели ООП

1) Модель **объекта**
2) Модель **класса** (частный вид **абстрактного типа данных**)
3) Модель **наследования**
4) Модель **инкапсуляции**
5) Модель **полиморфизма**
6) Модель **обмена сообщениями**

2 Объекты, классы объектов и их реализация в языке C++

2.1 Концептуальная модель объекта

2.1.1 Определение объекта

Определение 1

Объект *— сущность, которая:*

- ***отличима*** *от себе подобных;*
- ***содержит*** *в себе* ***атрибуты*** *и* ***методы;***
- *обладает* ***состоянияем****, задаваемым атрибутами;*
- *обладает* ***поведением****, задаваемым методами;*
- ***динамическая*** *по природе.*

Атрибут *— сущность, отражающая какое-либо свойство объекта и имеющая значение.*

Метод *— сущность, описывающая действие, которое можно совершить с объектом.*

Поведение *— последовательность действий, которая может быть выполнена с объектом.*

Динамическая по природе *— может быть создана или уничтожена*

Состояние *— совокупность значений атрибутов.*

Отличимость *— объект может идентифицироваться по:*

- *имени;*
- *значению атрибута или совокупности значений атрибута.*

2.1.2 Методы

1) Каждый объект может иметь произвольное количество методов.
2) Каждый метод обязательно имеет следующую структуру:
   а) **селектор** (средство отличения одного метода от другого) — например, имя;
   б) указание на объект, которому он принадлежит;
   в) возможно, параметры.
3) Метод при выполнении своего действия может использовать атрибуты.
4) Метод может использовать другие методы этого же объекта *без* указания имени объекта, для котрого вызываются другие методы. (Если других объектов — имя указывать надо).
5) Метод объекта имеет право изменять атрибуты.

6) Результатом выполнения метода может быть изменение состояния объекта.
7) Изменение состояния зависит от назначения метода, состояния до вызова метода, значений параметров метода.
8) Метод, вызываемый вне объекта, которому он принадлежит, вызывается с именем объекта.
9) В объекте могут быть специальные методы, предназначенные для передачи сообщений. Результатом вызова их является прием либо передача сообщений.
10) Результатом выполнения метода может быть посылка сообщения.

2.2 Выделение объектов предметной области

1) Рассмотрим пример. Представление доски в аудитории 5143 в виде объекта

ПР1. Выбираем, что рассматривать:

ПР1.1 центральную часть реальной доски в аудитории — *неинтересно* (что в ней такого? прямоугольник как прямоугольник)

ПР1.2 полный комплект досок — *длинно* и *трудно*

ПР1.3 центральную доску со ствоками (2 штуки) — *существенно*. Створки можно открывать и закрывать. При желании мы потом сможем расширить модель на доски с большим количеством створок.

Вывод: принимаем последний вариант.

ПР2. Состав атрибутов:

- левая створка
- правая створка
- центральная часть

Предполагаем, что левая и правая створка прикреплены к центральной части.

ПР3. Состав методов:

- повесить доску; створки закрыты
- снять
- написать на доске
- стереть с доски
- закрыть обе створки
- открыть обе створки
- закрыть левую
- закрыть правую
- открыть правую
- открыть левую

ПР4. Из ПР3 убрать «закрыть обе створки» и «открыть обе створки». *Обоснование*: можно реализовать с помощью методов открытия и закрытия отдельных створок

2) Рассмотрение объекта «доска 5143» привело к тому, что надо упорядочить процесс выделения объекта. **Методика выделения объекта**:
   а) Основные части методики — шаги, приводящие к **ОО-модели**.
   б) Выделяем проблемные сущности решаемой задачи. *Например*: сущность «доска 5143».
   в) Представляем «наглядно» свойства сущностей проблемной задачи. *Например*:
      1. доска 5143
      2. левая створка
      3. правая створка
      4. маленькую доску слева от основной доски не рассматриваем
      5. добавляем основное поле (до этого момента мы о нем не вспомнили)
   г) Используем наглядные формы представления сущностей, их отношений и действий. *Например*:

левая створка	основное поле	правая створка

д) Определяем действия с сущностями задачи. *Например*:
   1. Доска 5143
      (i) повесить
      (ii) снять
      (iii)написать на доске
      (iv)стереть с доски
   2. Основное поле
      (i) написать в основном поле
      (ii) стереть с основного поля
   3. Створка (и левая, и правая)
      (i) открыть, если закрыта
      (ii) закрыть, если открыта
      (iii)написать на створке
      (iv)стереть со створки

е) Провести анализ полученных решений и сопоставить их друг с другом (анализ на взаимосвязь и определенность). *Например*: вопросы про объект «доска 5143»:
   1. Где написать, если мест несколько (на основном поле, на левой створке с внешней стороны (если она закрыта), на левой створке с внутренней стороны (если она открыта) и т.д.
   2. Когда все створки закрыты, есть ли щель между створками и основной частью? В реальности есть, но учитывать ли это в модели?
   3. Что значит «написать», если учитывать части доски?
   4. Что значит «открыто» и «закрыто»?
   5. Чем отличается левая створка от правой, кроме названия? [Ответ: они открываются в разные стороны: разная семантика (смысл) действий]

ж) Пересматриваем проектные решения по результатам анализа.

з) Разрабатываем математические модели сущностей, претендующих быть объектами. В них нужно отразить специфику предметной области. Модели могут быть избыточными — это лучше, чем недостаточными. *Например*: прямоугольник по трем точкам — это недостаточная модель (нужно хотя бы алгоритм вычисления четвертой точки, условие на нее — что-нибудь в этом роде). Прямоугольник по четырем точкам и четырем сторонам — модель избыточная, но по крайней мере, она соответствует предметной области (геометрии).

Задание 1

Самостоятельно разработать математические модели объектов:

- *доска 5143;*
- *левая створка;*
- *правая створка;*
- *основное поле.*

Путь решения:

- *Доска состоит из прямоугольников одинаковой высоты*
- *Правая верхняя граница левой створки = левая верхняя граница основного поля*
- *Левая верхняя граница правой створки = правая верхняя граница основного поля*
- *Длина любой створки равна половине длины основного поля (доска закрывается без щелей)*
- *Открыть — отразить створку относительно соответствующей стороны, совпадающей с ближайшей вертикальной стороной основного поля*

Вывод: математическая модель точно определяет действия и помогает формулировать пред- и постусловия.

3) *Выводы*:
   а) Выделение объектов в некоторой предметной области требует знания предметной области.
   б) При формировании объектов, выделении их свойств приходится рассматривать достаточно много вариантов и выбирать те, которые наилучшим образом соответствуют решаемой задаче.
   в) Для придания точного смысла выделяемым оъектьам необходимо использовать математические модели.
   г) Выделение объектов и их свойств требует наличия опыта, т.к. на каждом шаге требуется формировать варианты и выбирать их, предвидя последствия принимаемых решений.
   д) Выделение и описание объектов требует:
      1. **творческого подхода** при построении набора вариантов;
      2. **критического мышления** при отборе допустимых вариантов;
      3. **умения принимать решения**, предвидеть и оценивать последствия;
      4. **самостоятельности**.

2.3 Концептуальная модель класса объектов

1) В ООП одной из основных моделей является **класс**.
2) Есть 2 подхода к определению класса в ООП:
   а) перечисление совокупности объектов
   б) описание набора атрибутов и методов
3) Перечисление совокупности объектов

Определение 2

Классом объектов *является совокупность объектов, обладающих одинаковым набором атрибутов и методов.*

4) Описание набора атрибутов и методов

Определение 3

Классом объектов *является модель (описание) набора атрибутов и методов, которыми должен обладать каждый объект класса.*

5) Определение 2 всегда порождает набор объектов, а определение 3 не подразумевает существования объектов. *Вывод*: среди методов должны быть методы создания и уничтожения объектов.
6) Основные свойства класса:
   а) идентифицируется именем;
   б) для **класса** и **объекта** определено отношение «быть объектом класса»;
   в) класс является **моделью** для создания объектов класса;
   г) класс содержит методы для создания объектов класса. Как правило, класс имеет несколько методов для создания объектов с разными свойствами (значениями атрибутов) и в разных условиях их появления.
7) Класс обязательно содержит метод для разрушения объектов класса. Как правило, существует только один такой метод.
8) Определение 3 всегда позволяет построить объекты с заданными атрибутами и методами, т.е., получить класс в смысле определения 2.

2.4 Инкапсуляция атрибутов и методов в классе объектов

2.4.1 Понятие инкапсуляции

1) **Инкапсуляцией** в ООП называется *сосредоточение* и *сокрытие* информации (атрибутов) и методов в **классе** объектов [если использовать определение 3] или в **объекте** [если использовать определение 2].

2) Существует 3 группы языков программирования:
   а) инкапсуляция в классе;
   б) инкапсуляция в объекте;
   в) инкапсуляция и там, и там.
3) Инкапсуляция означает, что внутренняя реализация объекта или класса *скрыта* от пользователя. При этом пользователь знает об атрибутах и методах *только* то, что предоставил ему объект или класс.
4) Совокупность атрибутов и методов объекта или класса, предназначенные для применения пользователем, называются **интерфейсом**[1] класса.
5) *Целью* инкапсуляции является защита данных: ограничение доступа к атрибутам и методам объекта или класса со стороны пользователя. Ограничение направлено на предотвращение внесения изменений, не разрешенных объектом или классом.
6) Предметом (**уровнем**) инкапсуляции может быть как объект, так и класс.
7) **Прагматика** (выгода) от инкапсуляции состоит в следующем.
   а) Инкапсуляция позволяет вносить изменения в скрытые части объекта или класса без изменения интерфейса, т.е. без последствий для пользователя[2].
   б) Обеспечивает защиту от использования программистом действий с атрибутами или вызова методов, не разрешенных объектом класса.

Задание 2

Составить аннотированный перечень (список с краткой характеристикой) интернет-ресурсов, посвященных стандарту языка C++ и методам использования языка при реализации ОО-программ.

8) Инкапсуляция является развитием принципов модульного программирования, которое сосредотачивало модули (части) программы, необходимые для ее функционирования.
9) Инкапсуляция является развитием способов защиты переменных и процедур в строго типизированных языках программирования.

2.4.2 Инкапсуляция в концептуальной модели ООП

1) **Сосредоточение** реализуется размещением соответствующих сущностей в пределах объекта или класса.
2) **Сокрытие** атрибутов и методов реализуется введением **правил видимости** атрибутов и методов в объекте или классе.
3) **Правила видимости** в концпетуальной модели ООП задаются двумя утверждениями:
   а) Методы объекта или класса *видимы* для всех пользователей класса или объекта.
   б) Атрибуты объекта или класса видимы только для тел методов этого объекта или класса.
4) Утверждение а) может быть понято всегда однозначно.
5) Утверждение б) может быть понято в двух смыслах. Рассмотрим пример на псевдокоде:

Класс «Точка»
атрибуты: ***integer*** x
integer y
методы: переместить на Δx, Δy
move_$\Delta x\Delta y$(Δx, Δy):
$x \leftarrow x + \Delta x$
$y \leftarrow y + \Delta y$

[1] Термин **интерфейс** в контексте ООП имеет несколько значений. Второе значение см. в разделе «Виртуальные функции».
[2] Имеется в виду не пользователь ОО-программы, а пользователь класса — скорее всего, программист.

переместить в точку (Точка **p**). **p** — в общем случае, другой объект того же класса «Точка»
moveToPt(Point *pt*):

два подхода:

«чистый» подход
введем методы
integer **get_*x*()**:
    return *x*
integer **get_*y*()**:
    return y
и воспользуемся ими для
доступа к значениям атрибутов x и y
скрытых объекте класса «Точка»:
    x ← *pt*.get_x()
    y ← *pt*.get_y()

«грязный» подход
разрешим получать значения
атрибутов другого объекта того
же класса:
    x ← *pt*.x
    x ← *pt*.y

Вывод: оба примера реализации функции «переместить в точку» в точности следуют утверждению б). Однако особенность ситуации состоит в том, что функция вызывается для одного объекта, а передаваться ей может совсем другой объект того же класса. Что видит метод — все объекты класса *или* только тот объект, для которого он вызван?
Разные ответы на этот вопрос порождают разные линии языков программирования, описываемые утверждениями.

а) В теле метода класса видимы атрибуты **любого** объекта класса, в том числе и того, для которого он вызван.
   Интерпретация: Это утверждение означает, что метод класса «знает» (= имеет прямой доступ к) любой атрибут из объектов данного класса, в частности, и того объекта, для которого вызван метод.
б) Метод класса «знает» **только тот** объект, для которого он вызван.

6) Утверждение 5а) задает уровень защиты на уровне класса.
7) Утверждение 5б) задает область видимости на уровне объекта.
8) Механизм защиты на уровне класса реализован в языке Simula. Механизм защиты на уровне объекта реализован в языке Smalltalk. Существуют целые группы языков между этими двумя крайностями. (*Например*, C++).

Задание 3

Какая из шести рассмотренных подмоделей ООП используется больше всего в связи с введением новой архитектуры компьютеров?

2.5 Классы и объекты в языке C++

Основные вопросы:

а) Какие возможности концептуальной модели ООП реализует язык C++?
б) Какие возможности C++ *расширяют* концептуальную модель ООП?
в) Какие свойства языка C++ позволяют разработать технологию работы программиста?

2.5.1 Возможности (конструкции) языка C++, реализующие концептуальную модель ООП

1) Класс C++ есть тип данных, задаваемый программистом.
2) Класс является составным типом данных и может иметь составляющие, называемые **членами** класса.
3) Члены класса разделены на 2 категории:
   а) **члены-данные** (атрибуты);
   б) **члены-функции** (методы).

4) Члены-данные *могут*[3] принимать значения, которые задают состояние объекта.
5) Членами-данными могут быть объекты других классов.
6) Члены-функции определяют набор действий, которые можно выполнить с объектами класса.
7) Члены-функции класса доступны любому методу этого класса.
8) Вне класса к классу не могут быть добавлены ни члены-функции, ни члены-данные.
9) Класс C++ снабжен *методами* создания объектов класса. Эти методы называются **конструкторами** класса.
10) Если программист не определил собственных средств создания класса, то язык *предоставляет* ему конструктор *по умолчанию*. **Конструктор, предоставляемый по умолчанию**, выделяет память под объект, но не задает никаких значений атрибутов (т.е., не выполняет инициализацию).
11) Конструктор, создавая объект, устанавливает отношение «быть объектом класса».
12) Класс в C++ снабжен средством уничтожения объекта класса. Этот метод называют **деструктором** класса. Деструктор может быть только один.
13) Исполнение деструктора ликвидирует отношение «быть объектом класса».
14) Если деструктор программистом не определен, вызывается деструктор, предоставляемый компилятором по умолчанию. Он выполняет только освобождение памяти, занятой под объект.

Задание 4

*Разработать **представительные** (представляют возможности) и **выразительные** («пройти мимо нельзя») примеры классов на C++, демонстрирующие возможности C++ по реализации концептуальной модели ООП в модели класса.*

2.5.2 Свойства языка C++, *развивающие* концептуальную модель ООП

1) Секции класса: ***private***, ***public***, ***protected***.
2) Класс C++ может иметь несколько секций (раделов), начинающихся с метки ***private***, ***public*** или ***protected***.
3) Каждая секция задает режим (возможности доступа) к именам класса, размещенным в этой секции.
4) Секция с меткой ***public*** означает, что члены класса, размещенные в этой секции, доступны всем пользователям класса.
5) Секция с меткой ***private*** включает члены класса, доступ к которым возможнен *только* со стороны *членов-функций* класса.
6) Секция с меткой ***protected*** предназначена для задания членов-данных, к которым организуется эффективный доступ со стороны методов классов-наследников.[4]
7) Члены класса могут быть распределены по секциям в соответствии с желанием программистов, с условием: каждый член класса может быть размещен в данной секции и во всех секциях вообще *только* один раз.
8) Класс языка C++ позволяет задать несколько средств создания объектов. Такие средства называются **конструкторами** (ctors). ctorы позволяют создавать объекты в соответствии с:
   а) заданными параметрами (информацией о членах-данных);
   б) условиями создания объекта;
   в) условиями использования объекта.
9) **Принципиально важно!** Программист может определить ctor, однозначно задающий состояние объекта (и даже несколько ctorов).

Если программист не определил ни одного ctora, то язык C++ предоставляет конструктор, называемый **default constructor** (**конструктор, предоставляемый по умолчанию**).

---

[3] Заметим: именно что *могут*, а не должны, как в модели ООП.
[4] См. п. 11) раздела 3.2 «Отношение наследования в языке C++».

Этот ctor предоставляется компилятором.

Еще есть ctor, **вызываемый по умолчанию** (без прямого участия программиста) — это любой конструктор без параметров или с параметрами, все из которых имеют значения по умолчанию.

10) ctor, предоставляемый по умолчанию, лишь выделяет память под объект, но ничего не инициализирует. Т.о, ctor, предоставляемый по умолчанию, создает **экземпляр класса** (**instance**), но не объект класса. (Экземпляр класса: состояние не определено; объект класса: состояние определено).
11) Класс C++ может иметь несколько ctor-ов, задаваемых программистом, в том числе — ctor, предоставляемый по умолчанию. Их количество определяется структурой членов-данных и той информацией, которая доступна при создании объекта. ctorы подчиняются правилам перегрузки функций класса: они различаются по **сигнатуре**.

Определение 4

Сигнатурой *метода в C++ называется совокупность имени метода и последовательности имен типов входных параметров этого метода. Возвращаемое значение в сигнатуру не входит.*

12) Язык C++ позволяет программисту задать средство разрушения объектов — **деструктор** (dtor). Деструктор может быть в классе только *один*. Деструктор *не* имеет параметров, поскольку может быть вызван и без непосредственного участия программиста.
13) Среди членов-данных м.б. **члены-данные, общие для всех объектов класса (ЧДОДВОКи**). Такие члены-данные существуют даже тогда, когда ни одного объекта класса не существует. Они могут применяться, *например*, для организации механизма **подсчета ссылок** (reference counting), в качестве отладочных переключателей и т.д.
14) В составе членов-функций класса м.б. **функции для обработки ЧДОДВОКов.**

2.5.3 Свойства класса, обеспечивающие эффективную технологию разработки ОО-программ

1) Количество секций и их содержание создает программист в соответствии со своими целями разработки ОО-программ.
2) C++ позволяет поместить члены-данные и члены-функции в секцию ***protected***, что расширяет возможности программиста по контролю доступа при использовании наследования.
3) В классе м.б. размещены определения других классов. Эти классы называют **вложенными**. Объекты этих классов могут быть использоваться только объектами класса, в который они вложены.
4) В классе м.б. заданы функции — не члены класса, имеющие прямой доступ ко всем членам класса. Эти функции называются **дружественными**. Дружественными м. б.:
   а) Члены-функции других классов.
   б) Все члены-функции одного или нескольких классов, не совпадающих с рассматриваемых классов. (В таком случае говорят о **дружественном классе**).
   в) Функции, не принадлежащие никакому классу.

   Дружественные функции не соответствуют концептуальной модели ООП, потому что они разрушают инкапсуляцию (по мнению Смольянинова). Это мнение ошибочно: http://www.parashift.com/c++-faq-lite/friends.html#faq-14.2. Но, следуя нашему «отцу и учителю», в дальнейшем дружественные функции рассматривать не будем.

2.6 Агрегация и композиция объектов и классов

Рассмотрим *примеры*.

– Как соотносятся объекты «банк» и «отделение банка»? Очевидно, если банк существует, то отделение тоже существует; если банка нет, то и отделения нет. Таким образом, отделение не может существовать само по себе.

– Как соотносятся объекты «точка» и «квадрат, левая верхняя вершина которого есть объект «точка»»? Рассмотрим факт существования точки на плоскости. Ясно, что точка может существовать и сама по себе, т.е., если квадрата не существует.

Рассуждения над этими примерами приводят к трем важным понятиям ООП:

– **ассоциация;**
– **агрегация;**
– **композиция.**

Целесообразность рассмотрения этих трех понятий определяется тем свойством [из концепции ООП], что атрибутами объектов могут быть объекты других классов.

1) Исходное положение:
   Атрибутами объекта или класса могут быть объекты *других* классов.
2) Вводятся понятия **ассоциации, агрегации** и **композиции** в модели ООП.

Определение 5

Ассоциация — это такое отношение между объектами, при котором один объект является частью другого.

3) Ассоциация порождает два понятия:
   а) **объект-часть;**
   б) **объект-целое**
   и устанавливает отношение «**быть частью** объекта».
4) *Вопрос*: Может ли объект-часть существовать без объекта-целого? Возможны два подхода, проиллюстрированные *примерами* в начале раздела.
   ООП отвечает: полезно и то, и другое.
5) ООП в соответствии с 4) предусмотрено 2 вида ассоциации:
   а) **агрегация;**
   б) **композиция.**

Определение 6

*Отношение ассоциации называется **агрегацией**, если объект-часть может существовать без объекта-целого.*

Определение 7

*Отношение ассоциации называется **композицией**, если объект-часть не может существовать без объекта-целого.*

6) В модели ООП понятия **ассоциации, агрегации** и **композиции**, определенные для объектов, переносятся на классы.

Определение 8

*Отношение **агрегации** для классов: класс, определяющий объект-часть, может существовать без класса, определяющего объект-целое.*

Определение 9

*Отношение **композиции** для классов: класс, определяющий объект-часть, не может существовать без класса, определяющего объект-целое.*

7) Композиция в языке C++ реализуется определением класса-части в составе класса-целого. Преимущества:
   – Скрытый класс-часть строго специализирован
   – Сокрытие информации

   Недостатки:
   – Воспользоваться может только класс-целое

2.7 Представление класса

1) Класс м. б. представлен в C++ одной из следующих четырех форм:
   а) определение класса;

б) объявление (проект) класса;
в) объявление имени класса;
г) шаблонный класс.

2) **Определение класса** содержит *всю* информацию об атрибутах и методах класса.
3) Определение класса м.б. представлено в нескольких файлах.
4) **Объявление** содержит всю информацию по использованию класса — список свойств (атрибутов) и **заголовки** членов-функций (методов). Либо у всех, либо у части функций в данном случае тела могут отсутствовать.

Определение 10

Заголовком *метода называется совокупность его сигнатуры и типа возвращаемого значения.*

5) **Объявление имени класса** нужно для написания программ, активно использующих указатели. Чтобы пользоваться указателями на экземпляр класса, в C++ не нужно знать объявление класса — достаточно только объявления имени.

2.8 Представление членов-данных класса на языке C++

Задание 5

Изучить самостоятельно. См. *п. 6.1 «Реализация членов-данных».*

2.9 Представление членов-функций класса на языке C++

Задание 6

Изучить самостоятельно. См. *п. 6.2 «Реализация членов-функций в языке C++. Указатель* ***this****: назначение и использование».*

Обратить внимание на 2 варианта: определение функций внутри определения класса либо вне его.

2.10 Защита членов-данных класса

Целью этого раздела является рассмотрение способов защиты членов-данных класса:

1) Защита членов-данных объектов класса от непредусмотренного использования.
2) Защита функций от непредусмотренного изменения переданных параметров.
3) Защита возвращаемых значений-объектов.
4) Варианты (уровни) защиты:
   а) защита на уровне значения;
   б) защита на уровне доступа.

2.10.1 Проблема защиты членов-данных класса

1) Подход:
   а) *Необходимость* защиты от преднамеренных и непреднамеренных изменений.
   б) Поиск *способов* защиты.
   в) Построение *приемов* (методов) защиты. [т.е., конкретизация способов защиты]
   г) *Оценка полезности* методов защиты.
2) С каких позиций мы будем рассматривать защиту? С позиции разработчика, который:
   а) знает решаемую задачу;
   б) знает требуемые *свойства* классов, *членов-данных*;
   в) хочет получить *корректную* и *надежную* программу:

Определение 11

Корректная *программа — программа, удовлетворяющая спецификации.*

Определение 12

Надежная *программа — программа, не меняющая своих свойств в процессе эксплуатации.*

г) знает, что класс может разрабатываться двумя способами:
   1. заданием проекта класса (здесь пользователь класса уже может видеть и пользоваться защитой);
   2. заданием определения класса.

3) *Необходимость* защиты членов-данных обусловлена:
   а) существованием членов-данных, которые по своей природе *не* должны изменять своего значения в течение всего периода существования.
   Например, квадрат с фиксированной левой верхней вершиной.
   б) существованием членов-функций, которые *не* должны изменять значения членов-данных, но из-за ошибок реализации могут это делать.
   Например, функции-селекторы (selectors, getters).
   в) стремлением защитить *члены-данные* от ***несанкционированного*** *доступа со стороны внешних пользователей* класса.
   г) возможностью передачи объекта функции в качестве входовыходного (INOUT) параметра или возврата объекта функцией с использованием своего имени.
4) В C++ члены-данные можно защитить одним из двух способов или их сочетаний, а именно:
   а) используя защиту на уровне доступа к члену-данному;
   б) используя защиту на уровне значения члена-данного.

2.10.2 Защита на уровне уровня доступа

1) Осуществляется размещением членов-данных в соответствующей секции класса. Тем самым определяется доступ со стороны внешнего пользователя класса.
2) *Пример*:

```
class Point {
public:     // К членам-данным: 1) определен доступ со стороны любого
   int x;   // пользователя класса; 2) действия с членами-данными определяет
   int y;   // программа, в которой класс используется, а не сам класс
};          // Никогда так не делайте. Дети Уганды этого вам не простят.
```

3) Концепция ООП требует, чтобы члены-данные размещались в закрытой секции. Для доступа к ним должны использоваться члены-функции, позволяющие использовать члены-данные извне класса. Эти члены-функции обычно позволяют получить и задать значения членов-данных класса.

Определение 13

Метод, задающий значение атрибута, называется ***модификатором*** *(mutator) или* ***сеттером*** *(setter). Обычно такой метод именуется set_имя_параметра или setИмяПараметра.*

Определение 14

Метод, возвращающий значение атрибута, называется ***селектором*** *(selector),* ***геттером*** *(getter) или* ***акцессором*** *(accessor). Обычно такой метод именуется get_имя_параметра или getИмяПараметра.*

4) *Пример*, соответствующий концепции ООП:

```
class Point {
private:    // К членам-данным: 1) определен доступ со стороны любого
   int x;   // пользователя класса; 2) действия с членами-данными определяет
   int y;   // класс, а не программа, в которой класс используется
public:     // Без использования средств, предоставленных классов, никакие
            // действия с членами-данными невозможны
```

```
    int getX() {
      return x;
    }
    int getY() {
       return y;
    }

    void setXY( int xx, int xy ) {
       x = xx;
       y = yy;
    }
};
```

2.10.3 Защита на уровне значений

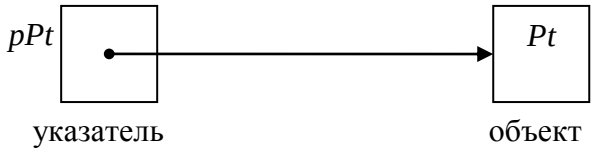
1) Защита на уровне значений при заданном уровне доступа защищает члены-данные от несанкционированного воздействия членов-функций класса.
2) Защита от членов-функций обусловлена тем фактом, что при релаизации членов-функций м.б. допущены ошибки *или* если члены-данные по своей природе *не* должны менять значения в течение своей жизни, а какие-то члены-функции пытаются их изменить.
3) В C++ средством защиты членых-данных на уровне значений является использование спецификатора (qualifier) ***const***.
   Например:
```
class SquareWithConstantPoint {
private:
   const Point leftTopPoint;
   // …
};
```
4) Член-данное, снабженный ***const***, сохраняет свое значение неизменным после инициализации в течение всего жизненого цикла (периода существования) объекта.
5) Значение, установленное для члена-данного со спецификатором ***const*** после инициализации, сохраняется неизменным и не может быть изменено ни одним членом-функцией, в том числе — и конструктором класса.
6) Инициализация, т.е., установка начального значения члена-данного, осуществляется при вызове конструктора перед выполнением тела конструктора. Она реализуется путем задания у конструктора **списка инициализации**. Защита осуществляется компилятором, не допускающим модификации константных членов-данных членами-функциями, в т.ч. и конструктором.
7) *Пример*.
```
class SquareWithConstantPoint {
   const Point leftTopPoint;
   unsigned int side;
public:
   void setLeftTopPoint( Point newLeftTop ) {
      leftTopPoint = newLeftTop;          // ошибка!
   }
  SquareWithConstantPoint( Point newLeftTop, unsigned int newSide ) {
      leftTop = newLeftTop;               // ошибка!
      side = newSide;
   }
   // …
};
```

Пример не откомпилируется, потому что метод setLeftTopPoint() и конструктор пытаются изменить член-данное leftTopPoint после его инициализации.

8) Если при задании члена-данного используется указатель, то возможны три варианта использования ***const***:

```
const Point *pPt;          // указатель на константную точку
Point *const pPt;          // константный указатель на точку
const Point *const pPt;    // константный указатель на константную точку
```

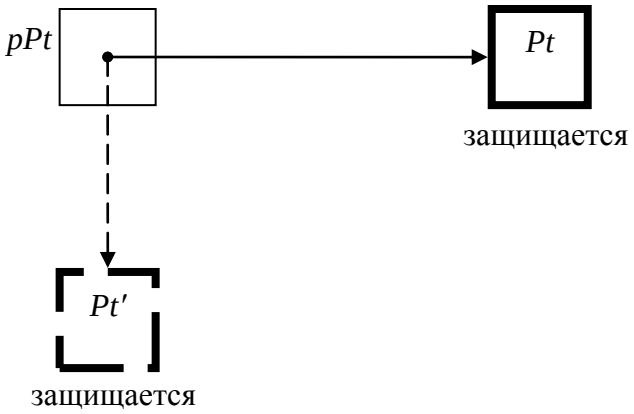
Эти варианты возможны, поскольку можно защищать как указатель на объект, так и сам объект:



Рассмотрим эти варианты на нашем примере (квадрат с фиксированной точкой).

9) Указатель на константную точку (защищается только объект)

```
const Point *pPt;
```



Семантика следующая:

- объект, на который указывает *pPt*, *не* может быть изменен;
- указатель *pPt может* быть изменен;
- изменение значения *pPt* означает, что точка-неизменная вершина стала другой. Указанная моделью цель (фиксированность точки) может быть достигнута только дисциплиной программирования[5].

Задание 7

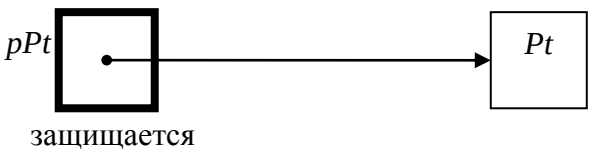
Теоретически[6] и экспериментально[7] ответить на следующий вопрос: Если член-данное задан конструкцией

```
const Point *pPt;
```

то можно ли, изменив указатель, изменить значение объекта?

10) Константный указатель на точку (защищается только указатель)

```
Point *const pPt;
```



Семантика следующая:

- объект, на который указывает *pPt*, *может* быть изменен;

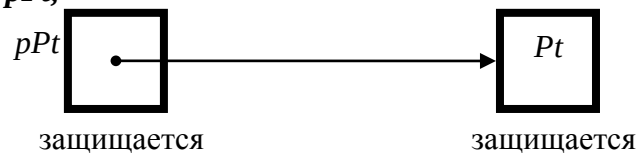
[5] Которая рано или поздно будет нарушена, со всеми вытекающими последствиями.
[6] Прибегнув к стандарту языка C++.
[7] Написав программу и проверив ее поведение для нескольких компиляторов.

– указатель *pPt не* может быть изменен.

Семантика *противоречит* модели нашего примера (в которой точка фиксированная).

11) Константный указатель на константную точку (защищается и указатель, и объект)

***const* Point *const pPt;**



Семантика следующая:

– объект, на который указывает *pPt*, *не* может быть изменен;

– указатель *pPt не* может быть изменен.

Такое использование полностью *соответствует* модели нашего примера.

12) «Вместо того, чтобы бороться с предметом, на который направлены «плохие» функции, сделаем функции «хорошими»».

Язык C++ для защиты значений членов-данных предусматривает специальные функции, называемые **константными**. Основное их свойство: они *не* могут изменять значения членов-даннных. Этот запрет реализуется компилятором языка следующим образом: всякая конструкция в теле функции, которая *может* изменить значение члена-данного, считается ошибочной.

13) Формат задания (*определения*) константной функции:

```
тип_возвр_знач имя_функции([список_параметров]) const {
  // тело функции
}
```

Формат объявления константной функции:

```
тип_возвр_знач имя_функции([список_параметров]) const;
```

Например:

```
class SquareWithConstantPoint {
  const Point leftTopPoint;
  unsigned int side;
public:
  const Point getLeftTopPoint() const {
    return leftTopPoint;
  }

  unsigned int getSide() const {
    return side;
  }
};
```

Вывод: все селекторные функции следует оформлять как константные.

Внимание: если константная функция вызывает неконстантную, компилятор может не заметить нарушения константности.

14) Член-данное может быть задан с использованием ссылки:

Point &leftTopPoint;

В C++ ссылки по своей природе константны: нельзя изменить адрес того участка памяти, где находится объект, на который ссылаются. Сам объект изменить можно.

Замечание.

Ссылки в C++ следует понимать как альтернативные имена для объекта. Именно поэтому для доступа к ним используется оператор ., а не ->. Ссылка — не указатель на объект, это сам объект, просто под другим, более удобным именем. То что обычно компиляторы реализуют ссылки с помощью указателей, еще не значит, что так следует понимать себе механизм работы ссылок.

Константная ссылка:

```
const Point &leftTopPoint;
```

соответственно, запрещает изменение объекта, на который ссылаются. *В некотором роде* (*см*. выше) константная ссылка аналогична константному указателю на константу — нельзя изменить объект по ссылке (а сама ссылка всегда неизменна).

Например:

```
class SquareWithConstantPoint {
   const Point &leftTopPoint;
   unsigned int side;
public:
   SquareWithConstantPoint( Point &newLeftTop, unsigned int newSide):
   leftTopPoint(newLeftTop) {
      // …
   }
   // …
};
```

Задание 8

Пусть проект класса и определение членов-функций расположены в разных файлах. Если мы хотим объявить константную функцию, в проекте пишем ***const****. А в определении функции надо?*

2.10.4 Защита объекта, передаваемого в функцию в качестве аргумента или возвращаемого функцией в качестве результата

15) Необходимость защиты аргумента вызвана тем, что члены-функции класса имеют право измнеять члены-данные передаваемого объекта.
16) Для организации защиты объекта, передаваемого в качестве аргумента, используется спецификатор ***const*** в составе параметра. *Например*:

```
class SquareWithConstantPoint {
   const Point &leftTopPoint;
   unsigned int side;
public:
   void isInSquare( const Point &pt ) {
      // Проверка наличия точки pt в квадрате не должна менять точку!
   }
   // …
};
```

17) Смысл задания параметра константным с помощью спецификатора в том, что аргумент, подставленный на место параметра, не может быть lvalue[8].
18) *Пример*.

```
class SquareWithPoint {
   Point leftTopPoint;
   unsigned int side;
public:
   void setLeftTopPoint( const Point newLeftTop ) {
      leftTopPoint = newLeftTop;
   }
// …
};
```

[8] lvalue — все, что может стоять в левой части оператора присваивания и аналогичных ему конструкций. Иными словами — все, что обладает именем.

Вопрос: можно ли в теле функции setLeftTopPoint() вызывать для leftTopPoint члены-функции, принадлежащие newLeftTop?

Вопрос: можно ли получить тот же результат следующим образом:

```
void setLeftTopPoint( const Point newLeftTop ) {
  leftTopPoint.setX( newLeftTop.getX() );
  leftTopPoint.setY( newLeftTop.getY() );
}
```

Ответ на оба вопроса: да, можно, но только если функции объявлены как константные. Функции, меняющие состояние newLeftTop, вызвать нельзя.

Задание 9

*Теоретически и экспериментально ответить на вопрос: в чем заключается различие (если оно есть) между передачей в функцию параметра по значению и использованием спецификатора **const** для этого параметра?*

19) Защита возвращаемого значения членов-функций осуществляется с использованием спецификатора ***const***. При этом ***const*** ставится перед типом возвращаемого значения:

```
const Point getLeftTopPoint() {
  return leftTopPoint;
}
```

Смысл использования ***const*** для возвращаемых значений состоит в том, что возвращаемый объект не может быть lvalue.

Задание 10

*Теоретически и экспериментально ответить на вопрос: если некоторый объект возвращается функцией в качестве **const**-объекта, можно ли для этого объекта вызывать функции-селекторы? А функции-модификаторы?*

Задание 11

Рассмотрим программу

```
class SquareWithConstantPoint {
  const Point &leftTopPoint;
public:
  Point &getLeftTopPoint() {
    return leftTopPoint;
  }
};
```

Не поймет ли компилятор, что функция getLeftTopPoint() может попытаться изменить значение члена-данного leftTopPoint, который объявлен константным?

Для ответа необходимо внимательно изучить стандарт C++ и выполнить вычислительный эксперимент (т.к., скорее всего, поведение будет зависеть от компилятора).

*Гораздо легче этого **не** делать и написать заведомо корректно:*

```
class SquareWithConstantPoint {
  const Point &leftTopPoint;
public:
  const Point &getLeftTopPoint() const {
    return leftTopPoint;
  }
};
```

Здесь:

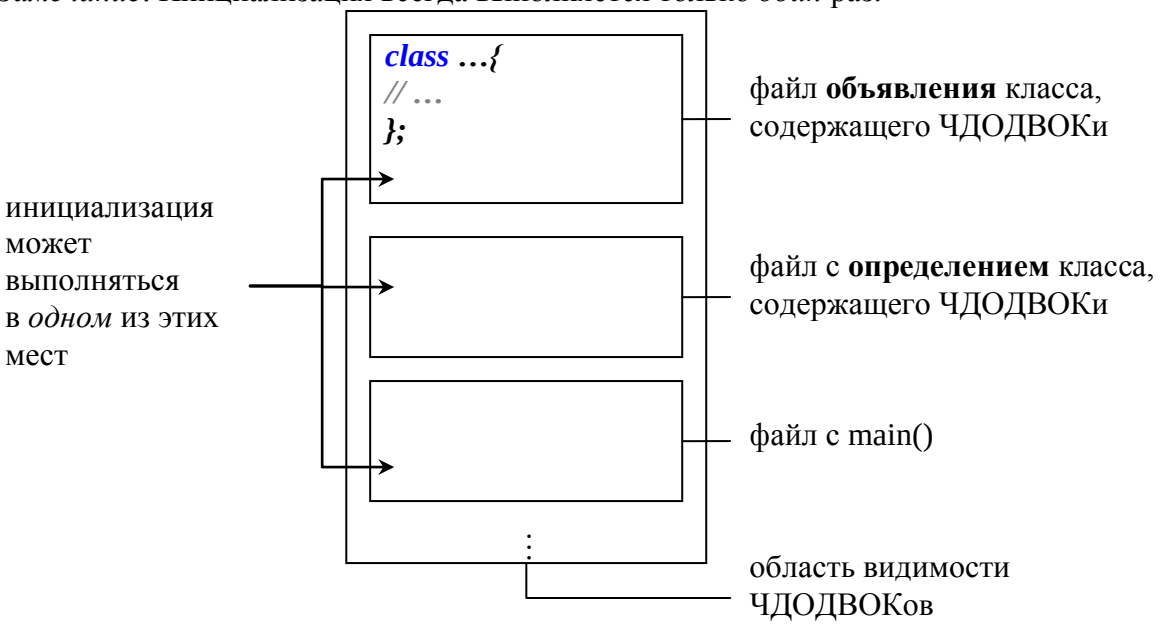
1. *Для защиты leftTopPoint в теле функции функция getLeftTopPoint() объявлена константной.*
2. *Для защиты возвращаемого значения оно объявлено константным.*

2.11 Реализация членов-данных, общих для всех объектов класса (ЧДОДВОКов) и членов-функций для их обработки

1) Концептуальная модель объекта предусматривет наличие атрибутов, общих для всех объектов класса, но не определяет способа их реализации.
2) Члены-данные, общие для всех объектов класса, могут существовать тогда, когда объекты класса еще не созданы или уже все уничтожены.
3) *Вопросы*:
   а) Как объявить ЧДОДВОКи?
   б) Как и где инициализировать их?
   в) Какова область видимости ЧДОДВОКов?
   г) Как организовать их обработку?

2.11.1 Реализация и семантика (смысл) ЧДОДВОКов

4) ЧДОДВОКи реализуются в C++ объявлением[9] членов-данных с атрибутом ***static***. Слово «static» обусловлено применением для реализации таких членов **статической памяти**, выделяемой в начале работы программы, освобождаемой при завершении работы программы и не зависящей от того, создаются объекты или нет.
5) Объявления членов-данных с атрибутом static м.б. размещены в любой из секций ***public***, ***private***, ***protected***. Здесь и далее мы будем размещать их только в секции ***private*** для последовательной реализации модели ООП.
6) Область видимости ЧДОДВОКов составляет область видимости класса, в котором эти члены объявлены.
   Замечание. Инициализация всегда выполняется только *один* раз.



7) ЧДОДВОКи создаются в момент входа в **область определения класса**[10].
8) ЧДОДВОКи уничтожаются после выхода за область определения класса.
9) Объявление члена-данного с атрибутом ***static*** позволяет:
   а) задавать или использовать значение такого члена-данного даже тогда, когда ни одного объекта класса еще не существует;
   б) использовать такой член-данное в тех случаях, когда все объекты уже уничтожены.
10) Жизненный цикл ЧДОДВОКа

---

[9] Объявляются члены-данные в одной из секций класса.
Определяются (инициализируются) члены-данные с помощью списка инициализации.
[10] Область определения класса — там, где классом можно пользоваться.

11) Формы обращения к ЧДОДВОКу:

[пространство_имен::]класс::имя_ЧДОДВОКа [по квалифицированному имени]

объект_класса.имя_ЧДОДВОКа

ссылка_на_объект_класса.имя_ЧДОДВОКа

указатель_на_объект_класса->имя_ЧДОДВОКа

2.11.2 Инициализация ЧДОДВОКов

1) ЧДОДВОК может быть **инициализирован** (снабжен начальным значением) только один раз. Для выполнения инициализации используется запись вида

   тип_данных квалифицированное_имя_ЧДОДВОКа = начальное_значение;

   где квалифицированное имя имеет вид

   [пространство_имен::]класс::имя_ЧДОДВОКа

   Заметим, что «=» здесь не операция присваивания, а **операция инициализации.**
2) ЧДОДВОК можно инициализировать либо в файле, содержащем определение класса, либо в любом другом файле, находящимся в области определения класса.
3) Если инициализация осуществляется в файле определения класса, то это инициализирующее значение будет использовано во всех программах, использующих класс.
4) Если инициализация осуществляется в другом файле, инициализация будет зависеть от конкретной программы. *Пример*: отключение отладочного переключателя в одной программе и включение его в другой.

2.11.3 Функции для использования ЧДОДВОКов

1) Для действий с ЧДОДВОКамии С+ предусматривает специальный класс функций. Этот класс образуют функции, объявляемые с атрибутом ***static*****.** Они тоже размещаются в статической памяти. *Пример* определения:

```
class Point {
   // …
   static unsigned int counter;
 public:
   static unsigned int getCounter() {
      return counter;
   }
};
```

2) Статическая функция-член класса м.б. объявлена в классе, а определена как в классе, так и вне его (но в области действия класса).
3) Если такая функция определена в классе, то атрибут ***static*** обязательно ставится перед возвращаемым значением.
4) Если функция объявлена в классе с атрибутом ***static***, то в определении вне класса этот атрибут *не* используется.
5) Статическая функция-член класса не может обращаться в своем теле к нестатическим членам-данным и функциям. (*Технически*: у этой функции нет первого неявного параметра ***this***).
6) Статические функции-члены не могут быть объявлены константными.

7) К статическим функциям можно обращаться двояко: через объект класса (если он существует) и по квалифицированному имени (вне зависимости от существования объектов класса).
8) В классе не может существовать две функции-члена, одна из которых статическая, а другая нет, а остальное в заголовке функции совпадает. (Т.е., ***static*** не есть средство переопределения функций).

Задание 12

*Теоретически и экспериментально ответить на вопрос: может ли функция с атрибутом **static**, являющаяся членом класса, использовать члены-данные своего класса, и если да, то каким образом?*

Задание 13

Разработать на основе материалов выполнения л/р №2 представительные и выразительные примеры реализации ЧДОДВОКов и функций для их обработки.

Задание 14

*Разработать технологию тестирования и отладки классов объектов, основанную на использовании ЧДОДВОКов. Начать с выбора способа (**черный** или белый ящик).*

2.12 Создание объектов класса и задание их начальных состояний

1) Для создания объектов класса и задания их состояний в C++ предусмотрены специальные функции, называемые **конструкторами** (ctorами).
2) Существуют ctorы, предоставляемые компилятором по умолчанию[11].
   а) **Конструктор класса, предоставляемый по умолчанию (default ctor).**
   Это конструктор без параметров. Он просто выделяет память под создаваемый объект (поскольку структура класса компилятору всегда известна, это сделать можно — в отличие от инициализации атрибутов).
   б) **Конструктор поверхностного (побитового) копирования (shallow (bitwise) copy ctor).**
   Это конструктор с одним параметром — ***const Class &obj***. Он «дословно» (**побитово**) копирует переданный объект. Если объект содержит внутри себя указатели или ссылки на другие объекты, то объекты, на которые они ссылаются, не копируются. Поэтому такое копирование называют **поверхностным**[12].

   Использование ctorов, предоставляемых по умолчанию — **ужас, летящий на крыльях ночи** с точки зрения надежного программирования.

2.12.1 Свойства ctorов, предназначенные для реализации модели ООП

1) ctor класса является в C++ функцией-членом класса, которая возвращает **экземпляр** класса.
2) ctor класса устанавливает отношение «быть **экземляром** класса». (Именно экземпляром, а не объектом. Состояние объекта задано, а состояние экземпляра не определено).
3) ctor класса в C++ может (но *не* обязан):
   а) задавать значения *всех* членов-данных класса;
   б) задавать значения *некоторых* членов-данных класса;
   в) вообще *не* задавать значения членов-данных класса.

   Очевидно, конструкторов б) и в) следует избегать, поскольку они не создают объекта в смысле концепции ООП.

---

[11] Подробнее о них см. далее.
[12] В противовес ему, **глубокое** (deep) копирование создает копии всех объектов, на которые есть указатели или ссылки в копируемом объекте класса.

4) ctor, предоставляемый C++ по умолчанию[13], не задает значений членов-данных, а только выделяет память. Он не создает объекта в смысле концепции ООП.
5) ctor языка C++, который задает значения всех членов данных, создает объект в смысле ООП. Такой ctor устанавливает отношение «быть **объектом** класса».
6) ctory можно передать инфорацию тремя способами или их комбинацией:
   а) передача значений членов-данных с помощью параметров конструктора;
   б) прямое задание значений членов-данных в теле конструктора;
   в) передача конструктору объекта того же класса, у которого конструктор возьмет все или часть значений членов-данных;
   г) передача значений членов-данных в списке инициализации конструктора.

   Упорядочим по убыванию: простоты б), г), а), в); полезности а), в), г), б).
   Критерии сложности: глубокое копирование — сложнее всего (объекты-части тоже могут состоять из частей, и надо предусмотреть, чтобы они были скопированы).
   Критерии полезности:
   - полностью задает значения или нет? Принимаем, что *полностью*;
   - возможность изменения задания значений в программе.
7) ctorы в общем случае м.б. разделены на 3 группы по способу создания объекта (классификация *условная*):
   а) конструкторы **копирования** (точная копия переданного объекта);
   б) конструкторы **создания по заданным параметрам** (значениям атрибутов);
   в) конструкторы **преобразования** (преобразование объекта заданного типа в объект рассматриваемого класса).

Определение 15

Конструктором копирования *в языке C++ называется ctor с заголовком вида*

```
Class( const Class &obj );
```

либо

```
Class( Class &obj ); // не рекомендуется
```

8) В принципе, конструктор копирования может иметь и более 1 параметра, но тогда все параметры, начиная со 2-го, должны иметь значения по умолчанию. *Например*:
```
Point( const Point &src, int y0 = 0) {
  x = src.x;
  y = y0;
}
```
   Полный формат определения ctora *требует*, чтобы параметры со 2-го и далее имели значения по умолчанию, поскольку ctor копирования может быть вызван компилятором без участия программистом (и компилятор не «знает», как задать остальные параметры). Наличие одного обязательного параметра — ссылки на исходный объект класса позволяет компилятору определить, что это ctor копирования.
   Если ctor копирования определен программистом, компилятор не предоставляет ctora поверхностного копирования.

Определение 16

Конструктор преобразования — конструктор с заголовком вида

```
Class( const AnotherClass &anotherObj );
```

т.е. конструктор, первый параметр которого не является объектом класса, для которого создается конструктор либо ссылкой на объект такого класса.

9) Если требуется предотвратить использование конструктора вида
```
Class( const AnotherClass &anotherObj );
```

[13] Умалчивается факт вызова ctora и значения членов-данных.

для неявного (выполняемого компилятором, а не программистом) преобразования объектов класса AnotherClass в объекты класса Class, конструктор объявляют с ключевым словом ***explicit***.

10) Параметры ctora могут быть защищены при выполнении действий по созданию объекта. Это осуществляется использованием спецификатора ***const*** у соответствующего параметра.
11) На этапе создания константность возвращаемого конструктором значения не обеспечивается, т.к. она может быть обеспечена при использовании.
12) Конструктор не может иметь спецификатор ***static***.
13) Параметры ctora могут иметь значения по умолчанию, что позволяет создаваьт в пределах одной и той же программы объекты с различными состояниями.

2.12.2 Автоматический вызов ctora

1) Автоматический вызов ctora при объявлении переменных предусмотрен в C++ в четырех ситуациях:
   а) ***Point P;***
   б) ***Point Q = P;***
   в) ***Point Z = Point(8, 10);*** *// противоестественно, но так тоже можно*
   г) ***Point R(2, 3);***
2) Форма 1а) означает в C++ вызов ctora без параметров для создания объекта с именем *P*.
3) Форма 1б) означает в C++ вызов ctora копирования, который скопирует объект *P* в поле *Q*, выделенное ctorом без параметров.
4) Форма 1в) предусматривает вызов ctora с 2 параметрами, и на место *Z* будет помещен созданный объект (здесь вызовется ctor копирования).
5) Форма 1г) предусматривает вызов ctora с 2 параметрами; результат: *P* равен созданному объекту.
6) Предпочтительные формы вызова ctorов при объявлении переменных — это 1а) и 1г). Копирование может быть дорогим (если требуется осуществлять очень глубокое копирование «тяжеловесных» объектов), а чаще всего оно не требуется.
7) Вызов конструктора производится в 5 ситуациях:
   а) Объявление объекла, локального в блоке[14] (см. пункт 1)). Там 4 ситуации.
   б) Передача объекта в тело функции по значению. **Никогда так не делайте!** Только по ссылке или указатель. Лишнее копирование никому не нужно.

```
// ужас, летящий на крыльях ночи
void doPipets( PipetsHelper ppz ) {
   // ...
}
// нормально
void doPipets( const PipetsHelper &ppz ) {
   // ...
}
```

   в) Возврат объекта функцией в качестве возвращаемого значения. **Никогда так не делайте!** Ссылка лучше.

```
// ужас, летящий на крыльях ночи
PipetsHelper returnPipets() {
   // ...
   return ppz;
}
// нормально
```

[14] **Блок** — область текста программы, между { и }. Объекты, локальные в блоке, уничтожаются при выходе из блока (при достижении }).

```
const PipetsHelper &returnPipets() {
  // ...
  return ppz;
}
```

г) Объявление динамического объекта[15].

```
Pipets &ppz = new Pipets( 42 );
```

д) Инициализация массивов. *Правило*: если список инициализирующих значений неполны й, то элементы, для которых предоставлены инициализирующие значения, принимают их, а остальные инициализируются вызовом конструктора без параметров.

```
Pipets ppzArray[4] = { Pipets( 42 ), Pipets( 43 ) };
```

2.12.3 Проектирование ctorов

1) C++ позволяет задать для класса несколько ctorов.
2) Количество и виды ctorов, которые могут существовать в классе, зависит от нескольких обстоятельств.
   а) От наборов начальных состояний, которые должны иметь объекты.
   б) От способов создания объектов (по предоставленным параметрам, копирование, преобразование).
   в) От способов использования объекта (передача в функцию, …).
   г) Ситуации автоматического вызова конструкторов.
   д) Правила переопределения функций (однозначаная сигнатура ctora).
3) Рассмотрим процедуру проектирования ctora на *примерах*. Пусть есть 2 класса — точка и квадрат с фиксированной точкой.

```
class Point {
  double x;
  double y;
public:
  // ctorы
};
```

```
class SquareWithConstantPoint {
  Point &leftTopPoint;
  unsigned int side;
public:
  // ctorы
};
```

Пример 1.

1. Возможные значения состояний объекта класса Point:
   $x = 1.0, y = 2.0$
2. Вид ctora. Возможны следующие ctorы:
   –
   ```
   Point() {
     x = 1.0;
     y = 2.0;
   }
   ```
   –
   ```
   Point(): x( 1.0 ), y( 2.0 ) {
   {
   }
   ```
3. Способы использования объекта:
   – создание автономных объектов класса Point;
   – создание членов-данных класса Square.
4. Каждый из ctorов может быть вызван автоматически.
5. В классе Point оба ctora одновременно существовать не могут (*см*. правила переопределения).
6. Эффективность использования ctora.

---

[15] В отличие от объекта, локального в блоке, динамический объект создается оператором ***new***. Память, отведенная под него, освобождается оператором ***delete***. Автоматического освобождения памяти не производится. Существуют способы достичь с динамическими объектами того же, что происходит с локальными (например, ***auto_ptr*** из стандартной библиотеки C++).

- *Оставляем* вариант со списком инициализации.
- *Причины*:
  - идейное разделение во времени подготовки среды, в которой действует ctor, и непосредственно действий по подготовке к использованию объекта класса;
  - высвобождение тела ctora;
  - большая эффективность использования списка инициализации (предотвращение преждевременной пессимизации [Саттер])

Пример 2. Для тех, кому не нравятся ctorы без параметров

1. Возможные значения задаются параметрами. Имеются приоритетные значения атрибутов объекта класса Point:
   $$x = 1.0,\ y = 2.0$$
2. Возможны ctorы:
   - ***Point(double x_0 = 1.0, double y_0 = 2.0) {***
     x = x_0;
     y = y_0;
     }

   - ***Point(double x_0 = 1.0, double y_0 = 2.0): x(x_0), y(y_0) {***
     }
3. Способы использования объекта:
   - создание автономных объектов класса Point;
   - создание членов-данных класса Square.
4. Каждый конструктор может быть вызван автоматически.
5. Оба конструктора одновременно в классе находиться не могут.
6. Вариант со списком инициализации эффективнее.

Пример 3. Ослабление зависимости Square и Point

1. Приоритетные значения будем задавать в конструкторе класса Square. Это ослабляет зависимость класса Square от класса Point (что, если Point поменяет приоритетные значения, на которые мы закладывались? А здесь мы непосредственно их задаем). Пусть конструктор класса Point не имеет параметров по умолчанию:
   Point(double x_0, double y_0): x(x_0), y(y_0) {
   }
   Square(unsigned int newSide): leftTopPoint(1, 2), side(newSide) {
   }
2. Пусть конструктор класса Point имеет параметры по умолчанию. Тогда можно независимо менять приоритетные значения для автономной точки и точки в составе квадрата.
   Point(double x_0 = 0, double y_0 = 0): x(x_0), y(y_0) {
   }
   Square(unsigned int newSide): leftTopPoint(1, 2), side(newSide) {
   }
3. Пусть координаты верхней левой точки квадрата будет задавать пользователь класса:
   Square(double left, double top, unsigned int newSide):
   leftTopPoint(left, top), side(newSide) {
   }

Варианты 2) и 3) наиболее гибкие.

3 Системы классов, основанные на использовании наследования

3.1 Отношение наследования в модели ООП

Определение 17

Пусть B и D — классы. Говорят, что B и D связаны ***отношением наследования*** *и записывают D ⇾ B, если:*

1. *Класс D полностью заимствует из B все атрибуты, методы и способы их использования.*
2. *Класс D может добавить к полученным из B атрибутам, методам свои атрибуты, методы и способы их использования.*
3. *Объекты класса D могут вести себя точно так же, как объекты класса B.*

1) **Отношение наследования** основывается на идее **заимствования** одним классом *D* структуры и действий, определенных в другом классе *B*, и включения их в состав другого класса с сохранением всех возможностей, которые были в заимствованном классе, включая поведение.
   По сравнению с композицией, наследование заимствует класс (модель), а композиция — объект.
2) Пункт 3) в определении отношения наследования выражае важный момент. Отношение наследования — это отношение isA («является»). **Если нельзя сказать: «*D* является *B*», то применение наследования ни к чему хорошему не приведет!**
   Замечание. Композиция, соответственно, устанавливает отношение hasA («включает в себя»). **Если нельзя сказать: «*B* является частью *D* (*D* включает в себя *B*)», то применение композиции тоже ни к чему хорошему не приведет!**
3) Объект класса *D* может быть подставлен в те места, где используется объект класса *B* (**принцип подстановки**).
   Принцип подстановки можно применять, только если объекты класса *D* ведут себя точно так же, как объекты класса *B* (за исключением новых методов).
   С формальной точки зрения, предусловия методов класса *D* должны быть такими же сильными (ограничивающими) или более сильными, чем у тех же методов класса *B*, а постусловия — такими же или более слабыми. Тогда все свойства, доказуемые для объектов базового класса, справедливы для объектов производного. [Liskov substitution principle]
4) Объект класса *D* может добавить свои атрибуты и методы, а может и *не* добавлять (это вполне законно). Добавление может производиться разными способами.

Определение 18

Конкретные способы добавления атрибутов и методов в D по отношению к B называются ***сценариями наследования*****.**

5) Отношение наследования может быть построено одним из двух методов:
   а) **специализация** (дедукция);
   б) **обобщение** (индукция).
   Например:
   – кнопка является (isA) элементом управления, поскольку она имеет такие общие для всех элементов управления свойства, как положение, размер, текст и методы (нарисовать). Но кроме того, у нее имеется метод «нажать». (Для просто элемента управления он не имеет смысла). Тогда класс «Кнопка» можно получить **специализацией** класса «Элемент управления».
   – кнопка, флажок, переключатель имеют общие свойства (положение, размер, текст) и методы (нарисовать). Тогда класс «Элемент управления», обладающий этими свойствами и методами, можно получить с помощью **обобщения** классов «Кнопка», «Флажок», «Переключатель».
6) Наследование может быть **простое** и **множественное**.

Определение 19

*Наследование называют **простым**, если класс D связан отношением ⇀ только с одним базовым классом B; и **множественным**, если с несколькими: $D \rightharpoonup B_1$, …, $D \rightharpoonup B_n$.*

3.2 Отношение наследования в языке C++

1) Отношение наследования в языке C++ отличается от отношения наследования в модели ООП.

Определение 20

*Класс, получаемый в результате наследования, называется **производным** (derived), а класс, от которого он наследуется — **базовым** (base).*

2) *Вопросы*:
   а) Какой класс может быть базовым?
   (Класс м.б. задан определением, проектом, объявлением имени класса).
   б) Как задать отношение наследования?
3) В качестве базового может выступать либо класс, заданный определением, либо класс, заданный проектом. Класс, для которого задано только объявление имени, базовым быть не может.
4) Формат задания отношения наследования
   class имя_производного_класса: спецификатор_доступа имя_базового_класса
   // …
   где
   class имя_производного_класса: — объявление класса как производного,
   : — знак **операции наследования**,
   спецификатор_доступа ::= public | private | protected,
   спецификатор_доступа имя_базового_класса — **описатель базы**.
5) Спецификатор доступа может быть ***public***, ***private*** или ***protected***. Значение этих ключевых слов мы подробно рассмотрим позднее. Если спецификатор отсутствует, подразумевается ***private***. **Будьте осторожны!** Обычно нам нужен спецификатор ***public***, поскольку только при использовании его в C++ реализуется **правило подстановки**.
   Примеры:
   ```
   class Button: Control {            // закрытое наследование
   };
   class Button: private Control {    // закрытое наследование
   };
   class Button: public Control {     // открытое наследование, наше любимое
   };
   class Button: protected Control { // защищенное наследование
   };
   ```
6) При любом виде наследования производный класс наследует все члены-данные и члены-функции базового класса. Доступ к ним определяется спецификатором наследования и правилами C++.
7) Производный класс может добавлять свои члены-данные и члены-функции.
8) Область видимости производного класса вложена в область видимости базового класса. Таким образом, члены базового класса видны в производном. Возможность их использования определяется спецификатором доступа.
9) Производный класс может **переопределить** члены базового. (Члены производного класса будут вызваны по именам членов базового класса; внутри класса будут вызваны переопределенные варианты).
   Пример. **Внимание! Не принимайте этот пример как руководство к действию!** Члены-функции производного класса будут вызваны не во всех случаях! *См*. раздел 3.7 «Виртуальные функции (вф)».
   ```
   class Button: public Control {
   ```

```
    unsigned int drawTop;
    unsigned int drawLeft;
public:
    // …
    void moveToPoint( const Point &pt ) {
        // вызов метода базового класса
        Control::moveToPoint( pt );
       // корректировка координат левого верхнего угла
       // закрашиваемой области с учетом отступа
        unsigned int margin = getMargin();
        drawLeft = pt.getX() + margin;
        drawTop = pt.getY() + margin;
    }
};
```

10) **Важно! Открывает огромные возможности C++**
Наличие или возможность существования производного класса в C++ может привести к изменениям в структуре базового класса.
   – ***protected*** и ***private*** наследования — не меняет структуру базового класса, но некоторое вмешательство в использование базового класса есть (влияет на доступ к членам базового и производного классов);
   – виртуальные (***virtual***) функции.
11) Использование защищенной (***protected***) секции в базовом классе специально ориентировано на эффективное обеспечение наследования. *Смысл*: позволяет членам-функциям производного класса обращаться к членам-данным секции ***protected*** базового класса напрямую. При этом члены-функции других классов могут обращаться к членам-данным секции ***protected*** только с помощью соответствующих членов-функций базового или производного классов. К членам-данным секции ***private*** доступ у всех только с помощью соответствующих членов-функций.
12) Виртуальные функции. *virtual* (англ.) — воображаемый. В данном случае «воображается» тело функции. Наличие в базовом классе виртуальных функций обусловлено возможностью использования правила подстановки объектов производного класса на место объектов базового класса.
В C++ правило подстановки «работает» только при ***public*** наследовании и только при использовании указателей или ссылок. Соответственно, подставляются указатели/ссылки на объекты производного класса вместо указателей/ссылок на объекты базового класса. Подробнее *см*. раздел 3.7 «Виртуальные функции (вф)».

3.3 Управление доступом к членам базового класса в производном классе

1) Под доступом к членам базового класса будем понимать **прямой** (непосредственный) доступ к членам базового класса, осущетвляемый с использованием имени члена базового класса (имя объекта не указывается).
2) Для членов базового класса (бк) можно определить набор пользователей этих членов — те конструкции, которые будут их вызывать.
Пользователи членов бк:
   а) члены-функции бк, включая ctorы и dtorы;
   б) члены-функции производного класса (пк), включая ctorы и dtorы;
   в) функции вне бк;
   г) функции вне пк;
   д) функции, дружественные пк и бк.
Пункт д) *не* берем, поскольку дружественные функции нарушают концепцию ООП.
3) Доступ к членам бк определяется следующими тремя обстоятельствами:

а) **вложение** областей видимости: область видимости пк вложена в области видимости бк;
б) **размещение** членов бк в *секциях* своего класса;
в) **спецификатор** вида наследования.

Табл. 1 Режим **прямого** доступа к атрибутам бк со стороны функций пк

секции бк / наследование	***public***	***protected***	***private***
public	все функции пк, бк и вне их	как к членам секции ***protected*** бк	—
protected	как к членам секции ***protected*** бк	как к членам секции ***protected*** бк	—
private	как к членам секции ***private*** бк	как к членам секции ***protected*** бк	—

4) При любом виде наследования функции-члены пк не имеют доступа к закрытым членам бк.
5) Открытое (***public***) наследование сохраняет режим достуа к членам бк, установленный самим бк.
6) Защищенное (***protected***) наследование обеспечивает доступ к открытым и защищенным членам бк как к защищенным (т.е., только для членов-функций пк).
7) Закрытое (***private***) наследование превращает все члены бк в члены закрытой секции бк. *Применить* это можно следующим образом: пусть у нас есть класс, в котором есть большое количество не нужных нам функций (которые могут быть использованы пользователями пк во вред) и пара-тройка нужных. Тогда произведем закрытое наследование от этого класса, а затем откроем те, которые нужны, *восстановив уровень доступа* (*см*. п. 8).
   Это синтаксически более короткий вариант **композиции**. Закрытое наследование устанавливает отношение hasA, а не isA (в отличие от открытого наследования). То же самое можно сделать, создав в закрытой секции объект базового класса, и написав методы для его использования. Закрытое наследование позволяет записать это *чуть* более коротко. Вместе с тем, при закрытом наследовании можно:
   – получить доступ к защищенным (***protected***) членам бк, что иногда бывает полезно (читай: обычно вредно, но как временная мера…);
   – переопределить виртуальные функции бк.
   Закрытое наследование дает дополнительные возможности, но и заставляет принимать дополнительную ответственность. Используйте **композицию** там, где это возможно; **закрытое наследование** — там, где это неизбежно. (*см*. подробнее http://www.parashift.com/c++-faq-lite/private-inheritance.html)
8) Уровень доступа к члену бк со стороны пк, измененный спецификатором отношения наследования, м. б. восстановлен в пк до уровня доступа, который этот член имел в соответствующей секции бк. Формат:

using имя_базового_класса::имя_метода;

Пример:

```
class Control {
public:
   void moveToPoint( const Point &pt );
};
class Button: private Control {
   // …
public:
   // выход №1: определить собственную функцию, вызывающую функцию бк
   // с тем же именем
   void moveToPoint( const Point &pt ) {
      Control::moveToPoint( pt );
   }
   // выход №2: указать компилятору восстановить уровень доступа,
   // указанный в бк
```

```
    using Control::moveToPoint;
};
```

9) Объявление уровня доступа к члену бк в пк может только восстановить (а не изменить) уровень доступа, указанный для члена бк в бк.

Задание 15

Теоретически и экспериментально ответить на вопрос: Как восстановить уровень доступа к функциям-членам бк, если они имеют одинаковое имя и разные списки параметров, причем м. б. размещены в разных секциях бк?

3.4 Реализация сценариев наследования

Определение 21

Сценарием *наследования называется набор применяемых в пк способов использования бк.*

1) Наследование — инструмент *повторного* использования ранее созданных классов.
2) Существует 2 принципиально различных способа использования отношения наследования для создания новых классов.
3) Первый путь состоит в том, что пк создается без учета использования программ и фрагментов, написанных для бк.
4) Второй путь состоит в том, что учитывается возможность использования объектов пк на месте объектов бк.
5) В концепции ООП путь всегда второй.
6) Для построение реального сценария наследования можно управлять уровнем доступа к членам бк в пк.
7) Инструментами задания сценария наследования являются:
   а) спецификатор типа наследования;
   б) способ восстановления уровня доступа;
   в) переопределение нужных функций.
8) Типовые сценарии наследования перечислены в табл. 2.

Табл. 2 Типовые сценарии наследования

Содержание сценария			Реализация	
Использование членов БК в ПК	Переопределение членов БК в ПК	Использование уровня доступа на основе БК	Вид наследования	Дополнительно
всех	нет	да	открытое	нет
не использ.	всех	да	открытое	переопределение
части	части	да	открытое	разрешение видимости
части	нет	да	закрытое	восстановление доступа
не использ.	нет	нет	закрытое	нет

3.5 Создание объектов пк и задание их начальных состояний

9) Основные *вопросы*, на которые требуется получить ответы:
   а) Нужны ли конструкторы пк?
   б) Если они нужны, то каково их назначение?
   в) Должен ли программист определять ctor производного класса?
   г) Как связаны между собой ctorы пк и бк?
   д) Сколько ctorов пк можно сделать? И чем это определяется?
   е) Зависит ли число ctorов пк от числа ctorов бк?
   ж) Может ли ctor пк инициализировать подобъект бк, и если да, то как?
10) Создание объектов пк осуществляется ctorами пк.
11) ctor пк *должен* создать объект пк, и задать значения членов-данных пк.
12) ctor пк *должен* создать **подобъект** бк, входящий в состав объекта пк, и задать его начальное состояние.
13) Язык C++, в общем случае, не требует от программиста создания конструктора пк.

14) Если программист не определил свой конструктор пк, язык предоставляет по умолчанию свой конструктор, который только выделяет память и наделяет ее правилами системы типов.
15) В связи с этим *вопросы*:
    а) Какой будет (и будет ли?) вызван конструктор бк, если конструктор пк предоставлен компилятором по умолчанию?
    б) Если программист решил определить один или несколько конструкторов пк, сколько он может их определить?
    в) Как должны быть связаны конструкторы пк с имеющимися в бк?
16) Рассмотрим связь конструкторов пк и бк на *типовых* примерах.
17) Ситуации, в которых программист должен определить конструктор или конструкторы пк, определяются следующими факторами:
    а) *необходимость* инициализации объекта пк — в концепции ООП это *абсолютная* необходимость;
    б) структура пк;
    в) свойства конструкторов бк.
18) Источником ситуаций могут быть свойства конструкторов бк:
    а) бк *не* имеет собственных конструкторов, определенных программистом — это прямое *нарушение* концепции ООП;
    б) бк имеет конструкторы, которые могут быть вызваны без параметров;
    в) бк имеет конструкторы, которые не могут быть вызваны без параметров.
19) *Ситуация 1*. бк не имеет конструкторов, определенных программистом, пк — тоже. В пк нет собственных членов-данных. *Пример*:

```
class Point {
   double x;
   double y;
public:
   void setXY( double newX, double newY );
   void getXY( double &currentX, double &currentY );
};

// прыгающая точка
class JumpingPoint: public Point {
public:
   void jumpToPoint( const Point &destination );
};
```

    Вывод: JumpingPoint можно использовать только при условии корректного использования объекта destination класса Point. Пример гадкий. **Никогда так не делайте** — данные должны быть инициализированы. А то точка может прыгнуть в неизвестно куда, если destination не имеет состояния в момент вызова ф-ции jumpToPoint().
    Задача: путем совершенствования класса JumpingPoint обеспечить корректное использование объектов этого класса.
    Решение: удовлетворительное решение невозможно. Требуется совершенствовать класс Point.
20) *Ситуация 2*. бк имеет конструктор, определенный программистом, который может быть вызван без параметров. пк не имеет конструктора, определенного программистом. пк не имеет собственных членов-данных.
    В ситуации 1 добавим ctor в класс Point:

```
Point( double newX = 1.0, double newY = 2.0 ) : x( newX ), y( newY ) {
}
```

    Вывод: созданный объект бк всегда имеет состояние, но оно может не удовлетворять требованиям контекста содержания или применения.

21) *Ситуация 3*. бк имеет конструктор, определенный программистом, с параметрами, который не может быть вызван без параметров. пк не имеет конструктора, определенного программистом.
    В ситуации 1 добавим ctor в класс Point:
    Point(double newX, double newY) : x(newX), y(newY) {
    }
    Тогда при компиляции класса JumpingPoint возникнет *ошибка*. Конструктор, предоставляемый компилятором по умолчанию, не сможет найти конструктор бк, который может быть вызван без параметров. А параметры для конструктора, который не может быть вызван без параметров, компилятору неоткуда взять.
    Вывод: конструктор пк в этом случае *должен* вызывать существующий конструктор бк и передать ему необходимые параметры. Конструктор пк *должен* быть реализован программистом. *Например*:
    JumpingPoint(double newX, double newY) : Point(newX, newY) { }
    Рекомендация: для пк *всегда* разрабатывать конструктор или конструкторы со списком инициализации. Их должно быть, как минимум, *не меньше*, чем конструкторов базового класса (и они должны иметь те же сигнатуры). Иначе мы *рискуем* не предусмотреть в пк те ситуации использования, которые были допустимы для объектов базового класса.
22) *Ситуация 4*. бк имеет конструкторы, все из которых не могут быть вызваны без параметров. пк не имеет собственных членов-данных.
    Вывод: если в пк *предполагается* использовать возможности всех конструкторов бк, то *необходимо* разработать в пк конструкторы, каждый из которых соответствует конструктору бк и обеспечивает передачу конструктору бк требуемого им набора параметров.

Задание 16

Разработать выразительные и представительные примеры ко всем четырем ситуациям, описанным выше.

3.6 Разрушение объектов производного класса

Проработать самостоятельно.

1) При разрушении объектов пк деструкторы вызываются в порядке, обратном порядку конструирования пк. То есть, сначала вызывается деструктор пк, а затем — деструктор бк. (Если бы было иначе, то деструктор пк не мог бы использовать состояние бк при выполнении деинициализации).
2) Если деструктор не предоставлен программистом, он предоставляется компилятором. Тело предоставляемого компилятором деструктора пусто.
3) После вызова всех деструкторов осуществляется освобождение памяти, отведенной под объект пк. (В эту память входит и память, отведенная под подобъект бк).

3.7 Виртуальные функции (вф)

Вопросы:

а) Что такое **вф**?
б) Для чего они нужны?
в) Как определить вф?
г) Какие бывают вф?
д) Как использовать вф?

3.7.1 Назначение вф

Рассмотрим назначение вф на *примере*. Пусть пишется программа для манипуляции с коллекцией, включающей как объекты пк, так и объекты бк.

Задание: используя классы Control и Button, разработать программу, которая:

4) *Объявляет* коллекцию указателей на объекты класса Control.
5) *Заполняет* коллекцию указателями на объекты класса Control и Button в *произвольном* порядке.
6) *Выводит* имена классов, к которым принадлежат объекты, указатели на которые являются элементами коллекции.
7) *Корректно* уничтожает объекты, на которые указывают элементы коллекции.

Исходные данные (код классов Control и Button):

```
class Control {
public:
   std::ostream &display( std::ostream &os ) {
      return os << "Control" << std::endl;
   }
};
class Button: public Control {
public:
   std::ostream &display( std::ostream &os ) {
      return os << "Button" << std::endl;
   }
};
```

Этапы и способы разработки

Способ разработки:

1) Поэтапная реализация частей программы по мере получения необходимых средств (конструкций языка).
2) Структура программы будет задаваться **стратегическими** (многострочными) комментариями

```
/*
    ⋮
*/
```

3) Пояснения к выполняемым действиям будут задаваться **тактическими** (однострочными) комментариями

```
// …
```

4) В результате реализации каждой части программы будут определены свойства конструкций языка требующиеся для реализации этой части.
5) В тексте программы будут приведены только те части, которые реализуют содержание задания.
6) В программе будет четыре части.

Набросок программы (пишем то, что можем написать):

```
/*
  1: Объявление коллекции указателей на Control
*/
control_collection *controls = new control_collection();
/*
  2: Заполнение коллекции указателям на Control и Button
*/
                                …
/*
  3: Вывод имен классов объектов, на которые указывают
  элементы коллекции
*/
for ( control_iterator i = control_collection.begin(); i != control_collection.end(); ++i ) {
```

```
    // функция display должна подменяться
    (*i)->display( std::cout );
}
/*
  4: Уничтожение объектов, указатели на которые содержатся в коллекции
*/
```

…

Вывод: для реализации пункта 3 *нужна* подмена функции display (вызов Control::display() для указателей на объекты Control, но Button::display() для указателей на объекты типа Button). Подмена *возможна*, поскольку используются указатели. В языке С++ эту задачу выполняют **вф** («воображаемые» функции: «воображается» алгоритм из другого класса).

3.7.2 Задание вф

Определение 22

Виртуальной функцией *в С++ называется функция-член класса, которая в пределах иерархии наследования позволяет определить для своего имени различные алгоритмы, зависящие от класса объекта, которому принадлежит эта функция.*

1) Вф может быть только функция-член класса (дружественная функция — нет).
2) Вф задается с использованием служебного слова ***virtual***. Формат задания вф:

   virtual обычное_определение_или_объявление_функции[= 0]
3) Если вф определяется внутри класса, то слово ***virtual*** используется в ее определении.
4) Если вф объявляется в классе, а определяется вне его, то слово ***virtual*** используется только один раз — в объявлении функции.
5) При объявлении и определении вф в классе, она выступает в этом классе в качестве представителя одноименных функций из иерархии наследования.
6) Если в классе определена или объявлена вф, она используется во всех пк как представитель одноименной функции при условии, что такая функция в классе не определена. Таким образом, в нашем примере для реализации пункта 3 необходимо записать

   ```
   class Control {
   // …
   public:
      virtual std::ostream &display( std::ostream &os );
   // …
   };
   ```

   Если в классе Button не определить функцию display() как виртуальную, то в наследниках Button будет вызываться Control::display(). (Поскольку Button::display() не будет «представителем одноименной функции…»). **Обычно это не то, что требуется от класса Button.**
7) Поэтому формулируем *правило*: если функцию можно переопределить в производном классе, то ее следует объявить виртуальной.
8) Если мы хотим переопределить вф, в пк ее имя, тип возвращаемого значения и список параметров должны быть одинаковыми.

Задание 17

Экспериментально и теоретически проверить, нужно ли использовать в каждом в производном классе у функции, претендующей быть виртуальной, служебное слово ***virtual****.*

9) Вф в классе языка С++ могут вообще не задавать никакого действия.

Определение 23

Виртуальная функция, не задающая вообще никакого действия, называется ***чисто виртуальной****, а класс, содержащий такую функцию —* ***абстрактным****.*

10) Формат задания чисто виртуальной функции
```
virtual обычный_формат_задания_функции = 0;
```
где «=» имеет смысл знака создания чисто виртуальной функции.
Тела у такой функции *не* бывает!

11) Если функция объявленная как чисто виртуальная, то класс, в котором она находится, *не* может иметь объектов.
Чисто виртуальная функция используется в программе для выражения концепции построения иерархии наследования. *Например*:
```
class Control {
public:
   virtual void setText( const std::string &ctlText ) = 0;
};
class TextBoxBase: public Control {
// …
public:
   virtual void setText( const std::string &ctlText ) {
      Core::sendMessage( Core::Messages::SetEditText, Core::Params< ctlText > );
   }
};
class RichTextBox: public TextBoxBase {
// …
public:
   void setText( const std::string &ctlText ) {
         // Если текст в точности такой же, какой уже задан, то ничего делать
         // не надо
         if ( _textCompare( ctlText, _cachedText ) == 0 ) return;
         // Если текст содержит инструкции форматирования,
         // отобразить его как форматированный; иначе — как простой текст
         Core::sendMessage( Core::Messages::SetRichEditText, Core::Params< ctlText ,
           ( _isRTF( ctlText ) ? TextFormats::RichText : TextFormats::PlainText ) > );
         // Сохранить заданный текст
         _cachedText = ctlText;
   }
};
```
Вывод: чисто виртуальные функции полезны.

12) Достаточно только одной чисто виртуальной функции, чтобы класс был абстрактным.
13) Если в бк определена чисто виртуальная функция, то в пк имеются 2 возможности:
   а) определить свой вариант чисто виртуальной функции (без служебного слова ***virtual***);
   б) продолжить построение упорядоченной системы абстракций — ограничиться чисто виртуальной функцией из бк.
14) В пк м. б. заданы свои чисто виртуальные функции, не связанные с чисто виртуальными функциями бк.

3.7.3 Вызов вф

1) Вф могут быть вызваны с помощью:
   а) оператора селекции ., примененного к именованному объекту;
   б) оператора селекции ., примененного к ссылке на объект;
   в) оператора селекции ->, примененного к указателю на неименованный объект.
2) Вф для объекта, имеющего имя, может быть вызвана с помощью оператора селекции .. При этом вф должна быть объявлена и определена в классе, которому принадлежит именованный объект.

3) Особые свойства вф как члена-функции класса реализуются в C++ *только* с помощью указателей и ссылок.
4) Набор «потенциальных» вф, из которого можно вызывать функции при исполнении или обращении, состоит из:
   а) функции, определенной для класса, используемых в обращении;
   б) функции, которая переопределяет виртуальную функцию в производном классе.
5) Конкретная функция, которая вызывается, определяется классом, на объект которого указывает указатель или ссылка, используемая для обращения.
6) Это определяется во *время выполнения* (runtime) программы. Говорят, что при вызове вф осуществляется *позднее связывание* (late binding).

Определение 24

Если вызываемая функция определяется в момент компиляции, этот процесс называется ***ранним связыванием*** *(early binding) если в момент вызова —* ***поздним связыванием*** *(late binding).*

7) Механизм вф полезен для эффективного использования правила подстановки (*см*. раздел 3.1, пункт 3)).
8) *Вопрос*: Можно ли скрыть реализацию и сосредоточить ее на нужных уровнях иерархии?
   Ответ: вф позволяет сосредоточить реализацию на уровнях иерархии наследования и скрыть ее от вызывающей программы, использующей объекты иерархии.

3.7.4 Виртуальные деструкторы

1) Конструктор *не* может быть виртуальным.
2) Деструктор класса, который может выступать в роли базового, *обязан* быть виртуальным. Почему? Рассмотрим наш пример с проектированием программы, работающей с коллекцией. В 4-м пункте примера необходимо освободить память, занятую объектами типов Control и Button, указатели на которые помещены в коллекцию. Это делается оператором ***delete***:

```
for ( controls_iterator i = controls.begin(); i != controls.end(); ++i ) {
   delete (*i);
}
```

   Естественно, что нам *требуется*, чтобы при уничтожении объекта типа Control вызывался бы деструктор ~Control, а при уничтожении объекта типа Button — его деструктор ~Button. Таким образом, требуется обеспечить подмену: вызываться *должен* деструктор того класса, на объект которого показывает указатель. Иначе мы *рискуем* некорректным (неполным) освобождением памяти.
   Корректная подстановка может быть достигнута, если деструктор будет виртуальной функцией, т.е., будет объявлен в бк со служебным словом ***virtual***.
   Вопрос: а как же будет работать переопределение? Ведь ~Control и ~Button — методы с разными именами?
   Ответ: деструктор в классе всегда один, и его имя всегда ~имя_класса, поэтому найти его компилятору не составляет труда.
   Введем понятие **виртуального деструктора**.

Определение 25

Виртуальный деструктор *— деструктор, который при вызове подменяется на деструктор того класса, которому принадлежит уничтожаемый объект.*

Виртуальный деструктор имеет служебное слово ***virtual*** *в начале заголовка.*

3) Для уничтожения объектов используется в нашем примере используется оператор ***delete***, который выполняется по отношению к каждому элементу коллекции — указа-

телю на объект класса Control. При этом осуществляется желаемая подмена, поскольку деструктор класса Control объявлен как виртуальный.
Вывод: использование виртуального деструктора позволяет осуществить задуманную подмену при уничтожении объектов, на которые указывают элементы коллекции.

4) *Вопросы*:
   а) Нужно ли делать виртуальный деструктор в наследниках?
   б) Что должен делать виртуальный деструктор?
5) Программист *обязан* определить в базовом классе виртуальный деструктор. Тело у деструктора *должно* присутствовать (хотя оно *может* быть пустым).
6) Если в бк определен или объявлен виртуальный деструктор, то во всех пк обязан быть явно определен свой деструктор. При этом слово ***virtual*** при задании деструкторов во всех пк *не* употребляется.
7) Объявление или определение виртуального деструктора в бк автоматически делает виртуальными деструкторы *всех* пк.
8) При наличии в бк виртуального деструктора при вызове оператора ***delete*** для указателей на объекты пк будет использоваться деструктор того класса, на объект которого показывает указатель-операнд оператора ***delete***.
9) Программист *не* должен явно вызывать деструкторы исходного и базового классов.
10) Если в классе есть хотя бы одна вф, то *следует* определить и виртуальный деструктор.

Задание 18

Объяснить на примере, почему существует такая рекомендация.

3.7.5 Псевдовиртуальный конструктор (шаблон проектирования «Фабрика»)

Изложение отложено до появления свободного времени.

4 Полиморфизм

1) *Полиморфизм* (греч.) — много форм (много проявлений).

Определение 26

Полиморфизм *— набор способов, обеспечивающий возможность для имени функции иметь несколько алгоритмов, которые могут быть выполнены [в зависимости от ситуации].*

Определение 27

Моделью полиморфизма *называется согласованный набор способов, обеспечивающих множественность алгоритмов для одного имени функции.*

2) Полиморфизм является способом повторного использования ранее разработанного программного обеспечения.
3) Модель полиморфизма в ООП использует для своей реализации два понятия — **полиморфный объект** и **полиморфная функция**.
4) В ООП в качестве полиморфных объектов могут выступать две сущности:
   а) **полиморфная переменная**;
   б) **полиморфное выражение**.

Определение 28

Полиморфная переменная *— переменная в программе, которая может принимать значения разных типов.*

Определение 29

Полиморфное выражение *— выражение, которое может принимать значения разных типов.*

4.1 Виды полиморфизма

1) В ООП используются и реализуются 4 вида полиморфизма:
   а) **«чистый» («правильный») полиморфизм**;
   б) **полиморфизм включения**;
   в) **параметрический** полиморфизм;
   г) **полиморфизм специализации**.
2) Для всех видов полиморфизма будем пользоваться следующими свойствами:
   а) Количество и структура функций, с помощью которых реализуется вид.
   б) Наличие способов реализации полиморфных параметров.
   в) Наличие ограничений на полиморфизм параметров.
   г) Способ вызова полиморфных методов.
   д) Наличие и способ реализации в языке C++.
3) **«Чистый»** полиморфизм

Определение 30

Полиморфная функция *— функция, которая имеет хотя бы один* ***полиморфный параметр****.*

   а) «Чистый» полиморфизм: полиморфная функция одна, параметр один (возможно, принимающий разные типы). Функция обеспечивает действия, заданные классом, к которому относится переданный параметр.
   б) Один метод имеет одно тело, которое во еремя выполнения вызова интерпретируется по правилам того класса, которому принадлежит **аргумент** этого действия. **Аргумент** действия — *реальный* объект, в отличие от параметров, которые являются *формальными*.
   в) Параметр *должен* быть **полиморфным объектом**, а аргумент — *может* быть **полиморфным объектом**.
   г) Ограничение на полиморфные параметры состоит в том, что значения аргументов должны быть объектами классов из определенного списка.
   д) Вызов метода осуществляется с помощью имени этого метода. При этом на месте полиморфного параметра должен стоять полиморфный аргумент.
   е) В C++ «чистый» полиморфизм *не* реализован.
4) **Полиморфизм включения**:
   а) Методов может быть несколько. Все они имеют одно и то же имя и список параметров. Методы являются методами классов, связанных отношением наследования. Полиморфизм включения реализуется набором этих методов.
   б) Полиморфным параметром является сам объект одного из классов, входящих в иерархию наследования.
   в) Полиморфные объекты должны быть объектами класса, входящего в иерархию наследования.
   г) В C++ реализовано с использованием виртуальных функций, которые вызываются по указателю или ссылке на объекты.
5) **Параметрический** полиморфизм:
   а) В качестве параметров в методе используется неопределенный класс, который выставлен в качестве параметра, который может быть конкретизирован.
   б) В качестве полиморфного параметра используется неопределенный класс, который конкретизирован.
   в) Нужно, чтобы реализации имели одинаковый набор параметров, тела могут быть любыми.
   г) Вызов полиморфного метода осуществляется вызовом соответствующего метод с указанием конкретного класса, имя которого занимает имя неопределенного класса.
   д) В языке C++ реализовано с помощью шаблонных классов.

6) **Полиморфизм специализации**:
   а) Реализуется с помощью нескольких функций, количество которых не ограничено, но конечно.
   б) Каждый метод должен иметь списки параметров, различимые либо количеством, либо типами входных параметров.
   в) *Ограничение*: списки параметров не должны совпадать.
   г) Вызов по имени с заданием соответствующего списка аргументов.
   д) В C++ реализовано с помощью механизма переопределения функций (*например*, задание нескольких конструкторов).

Задание 19

Для всех видов полиморфизма подготовить представительные и выразительные примеры.

5 Множественное наследование и виртуальные классы

5.1 Множественное наследование

1) Модель ООП предусматривает,что при задании производного класса базовых классов может быть несколько.
2) Язык C++ позволяет задать для производного класса несколько базовых.
   Формат задания:
   class имя_производного: { [public | protected | private] имя_базы$_1$, […] }
3) Одно и то же имя бк нельзя повторять несколько раз при определении отношения наследования.
4) Область видимости пк при множественном наследовании включает область видимости *каждого* из бк, указанных при задании отношении наследования.
   Если в нескольких базовых классах используются члены с одинаковыми именами, то обращение к ним из производного класса возможно *только* по квалифицированному имени.
5) При создании объекта пк подобъекты бк создаются в том порядке, *в котором перечислены* имена классов при задании отношения наследования.
6) При разработке конструктора объекта пк со списком инициализации порядок перечисления имен бк в списке инициализации не учитывается. Подобъекты все равно будут инициализированы в том порядке, который указан при определении отношения наследования.
7) При открытом наследовании со стороны всех бк объект пк может использоваться как объект любого из бк, т.е., действует правило подстановки.

Задание 20

Теоретически и экспериментально проверить, насколько реализуется правило подстановки, если пк открыто наследует $(n-1)$ *бк, а один оставшийся — закрыто.*

5.1.1 Преимущества и недостатки множественного наследования

   а) *Плюсы*: Позволяет создавать новые классы надежным методом, используя ранее разработанные без их изменения. Это сокращает затраты на разработку класса и является формой повторного использования кода.
   б) *Минусы*: Увеличивает сложность создания и модификации системы классов. Увеличивает **связь** (coupling) между классами — а, как известно, изменения в базовом классе могут повлечь серьезные проблемы в производных (**проблема хрупкости базовых класов**, fragile base class problem).
8) Полезность множественного наследования оценивается специалистами *неоднозначно*. Оно серьезно усложняет поддержку иерархии классов.

Согласно исследованиям, проведенным в ≈ 1997 году, в 95% случаев применение множественного наследование *бесполезно*.
Якобы «ответ» на скептическое отношение к множественному наследованию: следует проектировать классы с учетом возможности множественного наследования! (Как будто нет других насущных проблем проектирования…)

9) Есть ли случаи, в которых множественное наследование оправданно? *Да*, их два.
   а) Для создания конкретного класса, объединяющего в себе свойства нескольких конкретных классов. — полезность *спорна*
   б) Для создания класса на основе **интерфейсов**, и конкретного класса, задающего некоторую реализацию этого интерфейса. Такой класс называется **миксином** (mix-in).

Определение 31
Интерфейсом *называется абстрактный класс, содержащий только чисто виртуальные функции.*

Пример. В пакете форм существует элемент управления «Полотно» (Canvas), на котором **можно рисовать**. Этот класс **использует стандартные средства захвата и освобождения системных ресурсов**, реализованные в данном пакете форм.

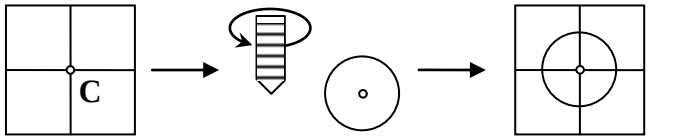
Концептуально:
– Полотно *является* (isA) элементом управления (Control).
– Полотно должно предоставлять функциональность по рисованию (IDrawable).
– Полотно должно обеспечить функциональность по освобождению системных ресурсов (IDisposable).
– Интерфейсы IDrawable и IDisposable задают, *что* можно сделать (наличие функций), но не определяют, *как* (функции чисто виртуальные). Можно сказать, что они определяют *контракт*, выполняемый классом Canvas, наследующим от них. При этом отношения isA между Canvas и IDrawable, IDisposable *не* устанавливается.

Пример кода:
```
class Canvas: public IDrawable, public IDisposable, public Control {
public:
   // реализация чисто виртуальной функции из IDrawable
   virtual void draw( Core::DeviceContext &dc );
   // реализация чисто виртуальной функции из IDisposable
   virtual void dispose();
   ~Canvas();
   // …
};
```

5.1.2 Многократное объявление базового класса в производном

Язык C++ *запрещает* многократное определение базового класса в определении отношения наследования, но *позволяет* объявить бк многократно через промежуточный класс.
Внимание! Обычно необходимость это делать говорит о том, что иерархия классов переусложнена!

Пример (бредовый) — автор Смольянинов А. В.
Пусть моделируется квадратная шайба с просверленным в центре круглым отверстием.

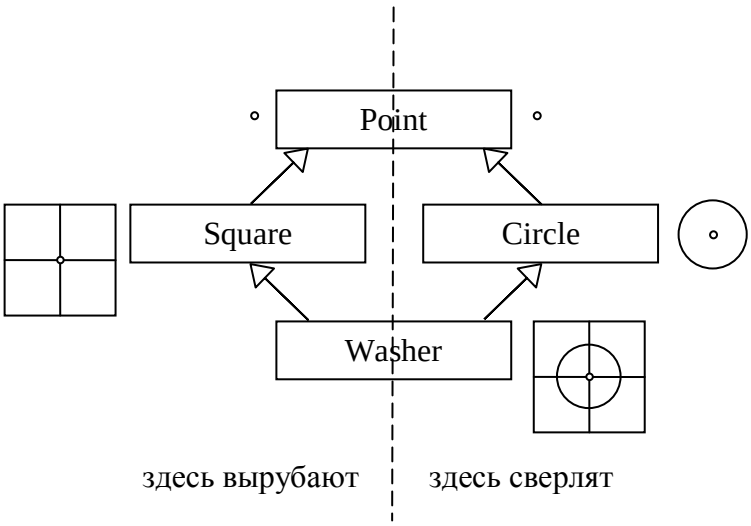


квадрат с подготовленным центром отверстия | сверление отверстия | шайба с круглым отверстием

Основные идеи моделирования с помощью ООП:

1) Квадратная заготовка изготавливается (а, значит, и моделируется) отдельно от сверления круглого отверстия.
2) За опорную точку в звготовке выбирается центр **C** пересечения диагоналей (и именно туда будет приставляться сверло).
3) Сверление должно осуществляться с центром в **C** .

Эти идеи выражает *диаграмма* UML:



Объявления классов:

```
class Point {
public:
   void moveToPoint( const Point &pt );
};

class Square: public Point {
public:
   void moveToPoint( const Point &pt );
};

class Circle: public Point {
public:
   void moveToPoint( const Point &pt );
};

class Washer: public Circle, public Square {
public:
   void skipToPoint( const Point &pt );
};
```

Выводы по проектированию:

1) Класс Washer имеет два подобъекта класса Point. Такой подход порождает более широкую модель. Возможностей в этой модели больше — можно сверлить не в центре, например, моделировать отклонение сверла. С другой стороны, для предотвращения ошибок в сверлении (чтобы получить то, что требуется по заданию — квадратную шайбу с круглым отверстием в центре), требуется следить, чтобы атрибуты подобъектов совпадали.
2) Клиенты класса Washer имеют доступ к методам moveToPoint обоих базовых классов.

Обращение к объектам класса:

```
Washer w; Point q;
w.moveToPoint( q );          // нельзя! не позволяет однозначно выбрать метод
w.skipToPoint( q );          // можно
w.Point::moveToPoint( q );   // можно
w.Square::moveToPoint( q );  // можно
w.Circle::moveToPoint( q );  // можно
```

5.2 Виртуальные классы

1) Если необходимо обеспечить при множественном наследовании только один экземпляр базового класса, C++ предоставляет такую функциональность.
2) Такая возможность реализуется в языке C++ объявлением базового класса как **виртуального**.

Определение 32

Виртуальный класс *— класс, который при множественном наследовании может существовать только в единственном экземпляре.*

3) Объявление виртуального класса осуществляется при задании отношения наследования:

```
class имя_производного: virtual public имя_виртуального, […]
```

Например:

```
class Circle: virtual public Point {
// …
}
```

4) При многократном объявлении базового класса в производном классе он представлен только одним объектом.
5) *Вопрос*: где производить инициализацию базового класса?
   Ответ: если виртуальный класс требует инициализации, то он инициализируется конструктором самого нижнего в иерархии наследования класса. В нашем примере (*см.* предыдущий пункт) это класс Washer.
6) Объекты виртуального класса создаются конструктором самого нижнего производного класса (потом базовых классов в порядке перечисления, потом — телом конструктора виртуального класса).

6 Дополнения (ответы на избранные вопросы к экзамену)

6.1 Реализация членов-данных

1) Конструкция становится членом-данным, если ее объявление размещается в какой-либо секции класса.
2) Член-данное представляет собой определение переменной, которое есть сочетание типа и имени.
   Формат члена-данного:

```
имя_типа имя_данного;
```

Например:

```
double x;          // абсцисса точки
double y;          // ордината точки
double r;          // длина радиус-вектора, r >= 0
double phi;        // полярный угол, \phi \in [0, 2\pi]
```

3) В качестве типов членов данных могут использоваться:
   а) предопределенный тип языка;
   б) класс, определенный пользователем;
   в) указатель на предопределенный тип или класс;
   г) ссылка на предопределенный тип или класс.
4) Использование указателя или ссылки не требует определение класса, достаточно только объявление класса.
5) При объявлении члена-данного, как члена класса инициализация невозможна кроме случая, когда инициализируется константный член-данное целого типа:

```
class Point {
   const int i = 8;          // можно!
   double x = 1.0;           // нельзя
};
```

6) В остальных случаях члены-данные могут быть инициализированы только конструктором.
7) Потенциальные пользователи члена-данного могут быть
   а) члены-функции своего класса;
   б) другие функции, не принадлежащие этому классу.
8) Члены-функции класса имеют прямой доступ к членам-данным своего класса. Это означает, что в телах этих функций обращение к члену-данному осуществляется только с использованием его имени.
9) Функции, не принадлежащие классу, в котором определен член-данное могут обращаться к этому данному только в том случае, если им разрешен доступ.
10) Уровень доступа определяет метка секции в которой определен член-данное:
   а) в открытой — разрешен доступ для любой функции;
   б) в защищенной — из функций, не принадлежащих этому классу, к нему имеет доступ только функции классов наследников;
   в) в закрытой — не имеет никакого доступа ф-ция, не принадлежащая этому классу.
11) Обращение к члену-данному, если это обращение разрешает уровень доступа, внешних функций осуществляется двумя способами:
   а) с использованием оператора селекции .;
   б) с использованием оператора селекции ->.

   Для реализации первого способа нужно знать имя, а второго — указатель на этот экземпляр.

6.2 Реализация членов-функций в языке C++. Указатель *this*: назначение и использование

6.2.1 Реализация членов-функций

1) Размещение, определение или объявление, функции в какой-либо секции класса делает эту функцию членом класса. *Например*:

```
class Point {
public:
   double getX() {
      return x_;
   }
};
```

2) Если в определении секции класса помещено объявление, то определение должно быть размещено отдельно.
Например:
```
class Point {
// …
public:
   double getX();
};
```
   а) Построение определения функции и связывание определения с объявлением.
   б) Класс не определение, а проект или прототип класса.
3) Если в определении класса использовано определение функции, то вне класса определение должно быть специальным образом оформлено. Это специальное оформление заключается в использовании формата оператора расширения области видимости.
Формат оператора расширения области видимости:
[пространство_имен::]имя_класса::
4) Формат определения функции члена-класса вне определения класса:
```
тип_возвр_знач [пространство_имен::]имя_класса::имя_функции( список_парам )
{
   // тело функции
}
```
Например:
```
double Point::getX() {
   return x_;
}
```
5) Члены-функции класса по своему назначению по отношению к пользователю класса делятся на 2 категории:
   а) внутренние функции (поддерживают работу класса);
   б) интерфейсные функции (реализуют действия предметной области, которые может выполнить пользователь с объектом класса).
6) Внутренние функции недоступны пользователю класса и предназначены для эффективной реализации интерфейсных функций.
Например: функции преобразования из полярной систему координат в декартову и обратно:
```
class Point {
   double x, y;
   void toPolar_( double &r, double &phi ) {
      r = sqrt( x * x + y * y );
      double raw_angle = atn( abs( y ) / abs( x ) );
      if ( x >= 0 && y >= 0 ) {
         // 1-й квадрант
         phi = raw_angle;
      } else if ( x <= 0 && y >= 0 ) {
         // 2-й квадрант
         phi = M_PI - raw_angle;
      } else if ( x <= 0 && y <= 0 ) {
         // 3-й квадрант
         phi = 3 * M_PI / 2 - raw_angle;
      } else if ( x >= 0 && y <= 0 ) {
         // 4-й квадрант
         phi = 2 * M_PI - raw_angle;
      }
   }
```

```
    void fromPolar_( double r, double phi ) {
      x = r * cos( phi );
      y = r * sin( phi );
    }
  public:
    // вращение относительно начала координат
    void rotateOverOrigin( double angle ) {
      double r,  double phi;
      // получим полярные координаты по x и y
      toPolar_( r, phi );
      // перейдем от повернутой точки в полярных к
      // повернутой точке в декартовых
      fromPolar_( r, phi + angle );
    }
  };
```

7) Интерфейсные функции предназначены для пользователей класса и задают набор действий, которые можно выполнить с членами класса. В предыдущем примере интерфейсная функция rotateOverOrigin() пользовалась для выполнения своих действий услугами внутренних функций toPolar_ и fromPolar_.
8) По возможности влиять на состояние экземпляра класса интерфейса функции делятся на 4 категории:
   а) селекторные (**селекторы**);
   б) модифицирующие (**модификаторы**);
   в) создающие объект класса (**конструкторы**);
   г) уничтожающие объект класса (**деструкторы**).
9) **Селекторы** предназначены для того, чтобы доставить пользователю класса значения атрибутов. Они не должны изменять атрибуты.
10) **Модификаторы** предназначены для задания значений атрибутов членов класса.
11) **Конструкторы** предназначены для создания экземпляров и возможно задания им начальных состояний.
12) **Деструктор** предназначен для разрушения экземпляра класса. Деструктор у класса один.

6.2.2 Указатель *this*: назначение и использование

1) Когда функция, принадлежащая классу, вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого она вызвана.
2) Этот указатель имеет фиксированное имя ***this***.
3) Имя ***this*** является служебным словом. Явно определить указатель ***this*** нельзя и не нужно.
4) В теле функции, вызванной для объекта, указатель ***this*** имеет смысл указания на самого себя.
   Например:

```
class Point {
  double x, y;
public:
   // Чисто для иллюстративных целей. Обычно гораздо лучше инициализатор.
  Point( double x = 0.0, double y = 0.0 ) {
    this->x = x;
    this->y = y;
  }
};
```

5) Использование указателя ***this*** — передача из объекта значения на самого себя для вставки этого указателя в массив или список. Как в предыдущем пункте — тоже можно, но применяется гораздо реже.
6) Также ***this*** используется при наследовании, чтобы вернуть указатель на объект базового класса.

6.3 Организация вычислений в концептуальной модели ООП

1) Обмен сообщениями в модели ООП является реализацией идеи *децентрализации* управления.
2) Объектно-ориентированная программа есть описание взаимодействия объектов.
3) Выполнение объектно-ориентированной программы состоит в обмене сообщениями между объектами.
4) Объекты существуют и реализуют внутреннее поведение. В какой-то момент времени объект передает сообщение одному или нескольким объектам.
5) Выполнение ОО программ заканчивается, когда все объекты перестают передавать сообщения. (*Замечание*. Практичнее было бы: когда все объекты перестают существовать. Впрочем, это влечет невозможность передачи сообщений.)

6.4 Проектирования обмена сообщениями в программах на языке C++

Определение 33

Сообщение *— указание другому объекту на выполнение некоторого действия. Сообщение, в общем случае, содержит три составляющие:*

1. *объект-получатель сообщения;*
2. ***селектор*** *сообщения — указание, какое действие должен выполнить объект-получатель;*
3. *параметры, необходимые для выполнения действия.*

6) Одной из форм передачи сообщения может быть следующее:

получатель.селектор(список_параметров);

7) Каждый объект, получивший сообщение, обязан выполнить действие, т.е. предоставить метод для выполнения действия.
8) Если объект-получатель не может предоставить метод для выполнения заданного действия, то он обязан выработать сообщение об ошибке.
9) Если у объекта есть метод, который выполняет заданное действие, то этот метод вызывается.
10) Объект-отправитель сообщает, как будет выполняться запрошенное действие, т.е. какой конкретный алгоритм будет использоваться для выполнения действия.
11) Действие, задаваемое в сообщении, не является фиксированным в том смысле, что алгоритм может зависеть от объекта, его выполняющего. Различные объекты, получающие одно и то же сообщение, могут выполнять различные действия. *Условно* это можно назвать «почтовым полиморфизмом».
12) В результате выполнения заданного действия могут быть изменены атрибуты получателя, или атрибуты класса, которому принадлежит объект. В результате исполнения запроса действия заданного сообщением, могут быть выработаны сообщения (отчет о выполнении, сообщение с результатом и т.п.).

Определение 34

Поведение *объекта — совокупность действий получателя сообщения.*

6.5 Реализация в ОО-программе на языке C++ почтового клиента для обмена сообщениями и его использование

Замечание. В этом и следующем пункте предполагается система Клиент→сервер→клиент.

1) Клиент — важнейшая часть почтовой системы, осуществляющая конечную обработку проблемных сообщений.
2) Клиент взаимодействует с другими клиентами через одни или несколько серверов, выполняя посылку и приём сообщений.
3) Клиент в общем случае имеет следующие функции:
   а) отправка сообщения:
      1. одному клиенту;
      2. [*необязательно*] всем клиентам;
   б) обработка принятого сообщения (выполнение действия, предписанного сообщением);
   в) регистрация на сервере (иначе как сервер пошлет сообщение клиенту?);
   г) разрегистрация (отмена регистрации) на сервере.

6.6 Реализация в ОО-программе на языке C++ почтового сервера для обмена сообщениями и его использование

1) Сервер — важнейшая часть почтовой системы, обеспечивающая взаимодействие клиентов путём обработки и передачи сообщений.
2) Сервер организует учет клиентов, их регистрацию и проблемное функционирование.
3) Сервер в общем случае имеет следующие функции:
   а) отправка сообщения:
      1. одному клиенту;
      2. [*необязательно*] всем клиентам;
   б) [*необязательно*] обработка принятого сообщения;
   в) перенаправление сообщения;
   г) регистрация клиента (иначе как клиент получит сообщение?);
   д) разрегистрация (отмена регистрации) клиента.

6.7 Проектирование по контракту

1) Основное назначение контракта — это определить строго условие использования его потребителем.
2) Условия использования задаются правами и обязанностями сторон.
3) Класс предоставляет пользователю класса некоторые возможности, определенные при заданных условиях и гарантирует организацию этих возможностей при выполнении этих условий.
4) Пользователь класса обязан выполнить условия, которые выдвигает класс для реализации возможностей.
5) Возможности класса и условия их реализации описываются с помощью утверждений.
6) Структуру контракта образуют
   а) предусловия для всех членов-функций;
   б) постусловия для всех членов-функций;
   в) инвариант для класса.

Определение 35

Предусловие post *для члена-функции класса — это [логическое] высказывание о свойстве параметров и членов-данных, которые необходимы для выполнения ее назначения.*

Определение 36

Постусловие pre *— это высказывание, описывающее свойство, возвращаемое функцией, значения, параметров и членов-данных, необходимых для реализации функции своего назначения.*

7) Инвариант класса определяет те свойства членов данных, которые присущи все объектам класса.

Определение 37 (рекурсивное)

Инвариант *класса* inv *— это утверждение, которое истинно тогда и только тогда, когда любой метод класса, вызванный*

1. *с аргументами, удовлетворяющими предусловию* pre *метода;*
2. *для объекта класса, состояние которого удовлетворяет инварианту класса* inv*;*

оставляет объект класса в состоянии, удовлетворяющем инварианту класса inv.
Иными словами, инвариант — это утверждение inv *такое, что*

$$\forall i : \{\mathrm{pre}_i \wedge \mathrm{inv}\}\ f_i\ \{\mathrm{post}_i \wedge \mathrm{inv}\}.$$

(где f_i *— члены-функции интерфейса класса).*

Определение 38

Контракт *класса — утверждение, включающее предусловия и постусловия для всех функций, а также инвариант класса:*

$$\mathrm{con} = \mathrm{inv} \bigwedge_i (\mathrm{pre}_i \wedge \mathrm{post}_i).$$

8) Контракт предназначен для проверки условий, порождаемых задачей и структурой программы, разработанной для ее решения. Эти условия могут быть нарушены программой.
9) Процесс составления контракта был подробно на практических занятиях и должен быть реализован при выполнении лабораторной работы.