**Билет 7**

1. Виртуальные методы классов. Назначение. Примеры

**Виртуальные методы** — это методы, которые могут быть переопределены в наследующих классах. Они позволяют обеспечить полиморфизм, то есть использовать объекты разных классов через общий интерфейс.

**Назначение**:

* Позволяют наследующим классам изменять или дополнять поведение метода базового класса.
* Обеспечивают динамическое связывание, где решение о том, какой метод вызывать, принимается во время выполнения программы.

**Пример**:

class Animal**:**

def speak**(**self**):**

print**(**"Animal speaks"**)**

class Dog**(**Animal**):**

def speak**(**self**):** # Переопределение метода

print**(**"Dog barks"**)**

class Cat**(**Animal**):**

def speak**(**self**):** # Переопределение метода

print**(**"Cat meows"**)**

# Создаем объекты

a **=** Animal**()**

d **=** Dog**()**

c **=** Cat**()**

# Все объекты могут вызывать метод speak()

a**.**speak**()** # Выведет**:** Animal speaks

d**.**speak**()** # Выведет**:** Dog barks

c**.**speak**()** # Выведет**:** Cat meows  
  
Здесь метод speak является виртуальным, и его поведение меняется в зависимости от типа объекта.

2. Используя объектно-ориентированный подход (допускается STL) написать программу, выполняющую следующие операции:   
а) Создание двумерного динамического вещественного массива, каждый элемент которого содержит вес дуги V(I,j), где I,j – индексы элемента. Причем для любой пары (I,j) существует путь L(I,j).

#include <iostream>

#include <vector>

**using** **namespace** std**;**

int main**()** **{**

int n **=** 5**;** // количество строк

int m **=** 5**;** // количество столбцов

// Создаем двумерный динамический массив, который будет хранить веса дуг

vector**<**vector**<**double**>>** V**(**n**,** vector**<**double**>(**m**));**

// Заполняем массив весами дуг (например, случайными числами)

**for** **(**int i **=** 0**;** i **<** n**;** **++**i**)** **{**

**for** **(**int j **=** 0**;** j **<** m**;** **++**j**)** **{**

V**[**i**][**j**]** **=** **(**i **+** j**)** **\*** 0.5**;** // Просто пример, можно использовать другие значения

**}**

**}**

// Выводим массив

**for** **(**int i **=** 0**;** i **<** n**;** **++**i**)** **{**

**for** **(**int j **=** 0**;** j **<** m**;** **++**j**)** **{**

cout **<<** "V(" **<<** i **<<** ", " **<<** j **<<** ") = " **<<** V**[**i**][**j**]** **<<** endl**;**

**}**

**}**

**return** 0**;**

**}**

б) решить задачу коммивояжера.  
Для решения задачи коммивояжера можно использовать алгоритм поиска с использованием динамического программирования или жадных методов. Простая версия решения:  
**Пример (Жадный алгоритм):**

#include <iostream>

#include <vector>

#include <cmath>

#include <limits>

**using** **namespace** std**;**

// Функция для вычисления расстояния между двумя точками (города)

double distance**(**int i**,** int j**)** **{**

// Это просто пример, обычно тут идет расчет реального расстояния

**return** abs**(**i **-** j**);**

**}**

int main**()** **{**

int n **=** 5**;** // количество городов

vector**<**vector**<**double**>>** dist**(**n**,** vector**<**double**>(**n**));**

// Заполняем массив расстояний

**for** **(**int i **=** 0**;** i **<** n**;** **++**i**)** **{**

**for** **(**int j **=** 0**;** j **<** n**;** **++**j**)** **{**

**if** **(**i **!=** j**)** **{**

dist**[**i**][**j**]** **=** distance**(**i**,** j**);** // Вычисляем расстояния между городами

**}** **else** **{**

dist**[**i**][**j**]** **=** 0**;** // Расстояние до самого себя равно нулю

**}**

**}**

**}**

// Простейшее решение задачи коммивояжера (жадный метод)

vector**<**bool**>** visited**(**n**,** **false);**

int start **=** 0**;**

visited**[**start**]** **=** **true;**

double total\_distance **=** 0**;**

int current\_city **=** start**;**

**for** **(**int i **=** 1**;** i **<** n**;** **++**i**)** **{**

double min\_dist **=** numeric\_limits**<**double**>::**infinity**();**

int next\_city **=** **-**1**;**

// Находим ближайший непосещенный город

**for** **(**int j **=** 0**;** j **<** n**;** **++**j**)** **{**

**if** **(!**visited**[**j**]** **&&** dist**[**current\_city**][**j**]** **<** min\_dist**)** **{**

min\_dist **=** dist**[**current\_city**][**j**];**

next\_city **=** j**;**

**}**

**}**

// Обновляем информацию

visited**[**next\_city**]** **=** **true;**

total\_distance **+=** min\_dist**;**

current\_city **=** next\_city**;**

**}**

// Закрытие пути (возвращение в начальный город)

total\_distance **+=** dist**[**current\_city**][**start**];**

cout **<<** "Общий путь: " **<<** total\_distance **<<** endl**;**

**return** 0**;**

**}**