4.12. Виртуальные функции. Объяснение логики выбора версии метода у наследника в случаях, когда функции виртуальные и когда нет. Логика выбора версий в случае, когда присутствуют как виртуальные, так и невиртуальные методы со слегка отличающимися типами параметров, разной константностью, разной приватностью и т. п.. Логика выбора версий в случае многоуровневого наследования и в случае множественного наследования.

Полиморфизм – один из главных столпов объектно-ориентированного программирования. Его суть заключается в том, что один фрагмент кода может работать с разными типами данных. (Например: операция плюс для целых чисел, для рациональных чисел, для матриц. Операция называется одинаково, но в зависимости от типа выполняются разные действия)

Виртуальная функция – это такая функция, что если к наследнику обратиться через ссылку на родителя, то выберется версия наследника.

```
#include <iostream>

struct Base {
    virtual void f() {std::cout << 1;}
    virtual void h() {std::cout << 1;}
    void g() {std::cout << 1;}
}

struct Derived: public Base {
    void f() {std::cout << 2;}
    virtual void h() {std::cout << 2;}
    void g() {std::cout << 2;}
}

struct Subderived: public Derived {
    void f() {std::cout << 3;}
}

int main() {
    Derived d;
    Base& b = d;
    b.f() // output: 2
    b.h() // output: 2
    b.g() // output: 1

    Subderived s;
    Base& b2 = s;
    b2.f() // output: 3 //Логика выбора версий в случае многоуровневого
}
```

При виртуальном наследовании при вызове метода вызывается более частная версия метода.

Выбор версии между виртуальной и невиртуальной

```
    struct Base {
        virtual void f() { cout << 1; }
    };
```

```
    struct Derived: public Base {
        void f() const { cout << 2; }//(1)
        virtual void f() const { cout << 2; } // (2)
    }

    int main() {
        Derived d;
        Base& b = d;
        b.f() // output: 1
    }
```

(1) - не виртуальный! потому что не полностью совпадает по сигнатуре. Т.е. небольшое изменение сигнатуры виртуальной функции приводит к тому, что она перестаёт быть виртуальной.

(2) - это всё ещё второй метод f, но уже виртуальный т.е. всё ещё не переопределяет Base f

Для виртуальных функций выбор происходит в Runtime, для не виртуальных в Compile time

Логика выбора версий в случае, когда присутствуют как виртуальные, так и невиртуальные методы со слегка отличающимися типами параметров, разной константностью, разной приватностью и т.п.:

Компилятор (в Compile time) для вызова выбирает версию функции, не смотря на виртуальность, а смотря на другие критерии (список аргументов, константность и т.д.). Поэтому среди виртуальных и невиртуальных функция выбирается функция вне зависимости от виртуальности. Если была выбрана виртуальная функция то в Runtime выбирается нужная версия этой виртуальной функции.

Логике множественного наследования.

Просто пример, лектор на эту тему особо не говорил.

```
struct Base1 {
    virtual void f() { cout << 1; }
}

struct Base2 {
    virtual void f() { cout << 2; }
}

struct Derived: public Base1, public Base2 {
    void f() {cout << 3;}
}

int main() {
    Derived d;
    Base1& b1 = d;
    b1.f() // output: 3
}
```

4.13. Чисто виртуальные функции, синтаксис определения, примеры использования. Понятие абстрактных классов. Виртуальный деструктор, особенности его определения и пример проблемы, возникающей в случае его отсутствия. Понятие RTTI, оператор typeid и особенности его использования.

```
class AbstractAnimal() {
    virtual int getAge(); // обычная виртуальная функция
    virtual void make_sound() = 0; // чисто виртуальная функция
}
```

При создании чистой виртуальной функции, вместо определения (написания тела) виртуальной функции, мы просто присваиваем ей значение 0.
Использование чистой виртуальной функции имеет два основных последствия. Во-первых, любой класс с одной и более чистыми виртуальными функциями становится абстрактным классом, объекты которого создавать нельзя.
Во-вторых, все дочерние классы абстрактного родительского класса должны переопределять все чистые виртуальные функции, в противном случае — они также будут считаться абстрактными классами.

Проблема виртуального деструктора (при отсутствии виртуальности).

```
struct Base {
    int* x = new int();
    ~Base() {
        delete x;
    }
}

struct Derived: public Base {
    int* y = new int();
    ~Derived() {
        delete y;
    }
}

int main() {
    Base* b = new Derived();
    delete b; // здесь вызовется деструктор для Base, а поле Derived::y не будет разрушено - утечка памяти
}
```

Проблема решается объявлением деструктора виртуальным методом.

RTTI (Run-time type information)

Динамическая идентификация типа данных (RTTI) — механизм, который позволяет определить тип данных переменной или объекта во время выполнения программы.

Если тип является полиморфным, то в compile time нельзя однозначно узнать, какую версию функции надо выбрать. Пример:

```
#include <iostream>
```

```
2
3  struct Base {
4      virtual void f() {
5          std::cout << 1;
6      }
7      virtual ~Base() = default;
8  }
9
10 struct Derived: public Base {
11     void f() override { //override - некоторое обязательство того, что функция
       виртуальная
12         std::cout << 2;
13     }
14 }
15
16 int main() {
17     int x;
18     std::cin >> x;
19     Base b;
20     Derived d;
21     Base& bb = x > 0 ? b : d;
22     bb.f();
23 }
```

int main() компилируется, если типы кастуются, что тут и происходит

Встроенный оператор typeid() - позволяет узнать тип выражения, известный компилятору (в Rantime). Реальный тип может быть другим, значит, компилятор вынужден в Runtime поддерживать информацию о типе. По этой же причине нельзя отследить проблему приватности и выбора версии.

```
1  struct Base {
2  }
3
4  struct Derived: public Base {
5  }
6
7  int main() {
8      Base b;
9      Derived& d1;
10     Derived& d2 = b;
11     cout << (typeid(b) == typeid(d1)) << endl; //false
12     cout << (typeid(b) == typeid(d2)) << endl; //true
13     cout << typeid(b).name() << endl; //3Base - некоторое строковое название
       структуры
14 }
```