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 типамиданных.

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;}
}

strcut Derived: public Base {
    void f() {std::cout << 2;}
}

int main() {
    Derived d;
    Base& b = d;
    b.f() // output: 2
}
```

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)

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

```
#include <iostream>

struct Base {
    virtual void f() {
        std::cout << 1;
    }
    virtual ~Base() = default;
}

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

int main() {
    int x;
    std::cin >> x;
    Base b;
    Derived d;
    Base& bb = x > 0 ? b : d;
    bb.f();
}
```

Оператор typeid() вычисляет тип объекта в runtime, возвращает объект типа std::type_info. Рекомендуется вместо него использовать decltype, так как обычно в C++ тип объекта известен заранее.