3.30 Мотивировка move-семантики. Какую проблему она решает? Пример ситуации, когда уместно использовать std::move. Пример правильной реализации move-конструктора и move-оператора присваивания для класса String.

Мотивировка move-семантики:

1. Когда мы хотим передать объект таким образом, мы вынуждены идти на одно "одноразовое"копирование

```
int main() {
    vector<string> v;
    v.push_back(string("abc"));
}
```

2. Далее есть реалокация при увеличении размера вектора, когда старый заполнился. А перекладывание - это снова $new(newarr+i)T(arr[i])$. Если в векторе лежали строки, то по факту - вектор просто хранил указатели на динамическую память каждой строки, тогда вопрос: зачем нам копировать все строки, если можно было просто переставить указатели (но напрямую так сделать нельзя)

3. Рассмотрим реализацию функции swap.

```
template <typename T>
void swap(T& x, T& y) {
    T tmp = x;
    x = y;
    y = t;
}
```

   В данном коде наблюдается тройное копирование в случае сложных объектов.

4. Пусть есть функция, результатом которой является новый объект типа T.

   (a) При вызове этой функции в другой части кода MyHeavyType object = createObject(); - будет все ок, так как произойдет Copy Elision (оптимизация компилятора).

   (b) Но если мы делаем f(createObject()), то мы не можем принять объект в f по значению, так как точно произойдет копирование (вызовется конструктор копирования). Если принять объект по константной ссылке, то мы не сможем менять его.

```
template <typename T>
T createObject(...) {
    //...
    return obj;
}

int main() {
    vector<string> v;
    v.push_back(string("abc"));
    //получаем лишнее копирование
    foo(createObject());
}
```

Пример использования:

```
template <typename T>
void swap(T& x, T& y) {
    T tmp = std::move(x);
    x = std::move(y);
    y = std::move(t);
}
```

После того, как объект мувают, все его параметры возращаются к дефолтным, например, размер мувнутой строки равен нулю. **К объекту, который мувнули, можно обращаться, гарантируется, что он останется в валидном состоянии.**

Если же объект мувнули и результат действия нигде не используется, то объект остаётся нетронутым:

```
int main() {
    string s("abc");
    std::move(s);
}
```

Пример правильной реализации move-конструктора и move-оператора присваивания для класса String

```
String(String&& s): sz(s.sz), s(s.str) {
    s.str = nullptr;
    s.sz = 0;
}

String& operator=(String&& s) {
    String temp = std::move(s);
    swap(temp);
    return *this;
}
```

3.31 Мотивировка умных указателей. Какую проблему они решают? Базовый синтаксис использования shared_ptr (как создать, как пользоваться в простейшем случае)

см. билет 4.28.