5.11 Реализация функции std::forward. Объяснение ее действия. Объяснение, почему принимаемый и возвращаемый типы именно такие. Почему в качестве принимаемого типа нельзя написать T&&?

Когда у нас в шаблонную функцию передается пременное число аргументов, и мы не знаем какие из них rvalue, а какие lvalue, можем воспользоваться функцией std::forward

```
template <typename ...Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Теперь все типы, кототые были переданы как lvalue будут иметь тип + &, а которые были переданы как rvalue - тип + &&. Пусть внутри функции f вызывается функция g, принимающая пакет аргументов, хотелось бы применить std::move к тем, которые являются rvalue и скорпировать остальные (к ним нельзя применять std::move, так как их передали в f не как rvalue, следовательно не ожидают удаления всей инфорации)

Реализация

```
template<typename T>
T&& forward(std::remove_Reference_t<T>& x) {
    return static_cast<T&&>(x);
}
```

Такая конструкция породит rvalue для объектов которые были изначальное отданы как rvalue и lvalue для всех остальных, это значит все аргументы проходят в функцию g с такими же видами value с какими нам их дали, как следствие сможем копировать только те объекты, которые нам изначально пришлось бы копировать. - **perfect forwarding** .

Именно perfect forwarding объясняет std::forward. Нам передаются аргументы, и мы не хотим постоянно тупо копировать (а в случае с rvalue создаются лишние копии). Тогда мы используем std::forward.

Полезная ссылка-пояснение. TL;DR - C++ 11 пытается сохранить rvalue как может.

5.12 Return Value Optimization (RVO), ее идея и пример, когда она возникает. Пример, когда небольшая модификация возвращаемого выражения приводит к исчезновению RVO (оператор += в BigInteger). Примеры, когда надо и когда не надо писать std::move после return. Понятие copy elision, идея этой оптимизации и пример ее возникновения. Условия, при которых она точно происходит, а также при которых может произойти, но не обязана.

Пускай мы перегружаем операторы.

Перегрузка арифметических операторов

В метод класса передаем только второй операнд, так как под левым операндом подразумевается *this.

```
struct Complex{
    double re = 0.0;
    double im = 0.0;

    Complex& operator +=(const Complex& z) {
        re += z.re;
        in += z.im;
        return *this;
    }
    Complex operator +(const Complex& z) {
        Complex copy = *this;
        copy += z;
        return copy;
    }

}

```

Оператор '+' должен создать копию объекта. Тогда если бы нам захотелось реализовать '+=' через '+' то на любое действие, даже при добавлении одного симовла к строке, создавалась бы ее копия и время работы оператора за счет копирования увеличивалось бы до $O(n)$, тогда как добавление одного символа к строке может быть реализовано за $O(1)$ - неэффективное решение.

Напоминание : возвращаем значение по ссылке, а не по указателю, так как ссылка ничего не весит.

Оператор '+' нужно определять вне класса, так как например для данной структуры при вызове

```
Complex c(2.0);
c += 1.0;
```

или при вызове

```
Complex c(2.0);
c + 1.0;
```

все сработает хорошо, компилятор сделает неявное преобразование double к Complex, однако следующий вызов

```
Complex c(2.0);
1.0 + c;
```

выдаст ошибку, так как мы определили оператор '+' только тогда, когда левым операндом является объект класса (*this).

При определении оператора вне функции и левый, и правый операнд будут равноправны, и компилятор сможет делать каст как левого, так и правого операнда - соответственно в оператор надо передавать два параметра. Тогда корректный код выглядит так:

```
struct Complex{
    double re = 0.0;
    double im = 0.0;

    Complex (const Complex&){
    }

    Complex& operator +=(const Complex& z) {
        re += z.re;
        in += z.im;
        return *this;
    }
}
```

```
14
15      Complex operator +(const Complex& a, const Complex& b) {
16              Complex copy = a; //Copy constructor definitely called
17              return copy += b;
18              //copy += b;
19              //return copy;
20      }
21
22      int main(){
23          Complex c(2.0);
24          Complex d (1.0,3.0);
25          Complex sum = c + d; //Copy constructor isn't called
26      }
27
```

В данном коде конструктор копирования вызовется 2 раза: в (16) строке - при создании copy, и в (17) - при возвращении результата (метод возвращает результат по значению, а значит создается копия результата). В (25) строке при присваивании суммы конструктор копирования не вызывается - происходит **copy elision** (появилось в C++11). Copy elision заключается в следующем: справа от оператора '=' после выполнения операции создался временный объект типа Complex, которым инициализируется левый операнд. Тогда компилятор не создается еще один временный объект для присваивания, а сразу считает получившийся временный объект нужным. Можно сократить количество копирований до одного с помощью **Return Value Optimization(RVO)** - если компилятор понимает, что в методе создается локальный объект и он же возвращается, то компилятор выделит память в том месте, где ожидается возвращание результата функции, таким образом убирая лишнее копирование. Заметим, что в незакомментированном коде это оптимизация не будет вызвана, хотя и возвращается объект, созданный в методе - компилятору не очевидно, что это тот же самый объект.

Очень полезная ссылка

Про std::move и return:

Иногда хочется написать std::move после return, чтобы соптимизировать, но так делать не всегда надо; из-за RVO это может ухудшить ситуацию. Например:

если возвращается не ссылочный тип, то из-за RVO компилятор может избежать перемещение И копирование, но std::move заставит делать перемещение, нет оптимизации. Поэтому если есть RVO, писать return std::move не надо, например:

Если вы имеете дело с функцией, осуществляющей возврат по значению, и возвращаете объект, привязанный к rvalue-ссылке или универсальной ссылке, вы захотите применять std::move или std::forward при возврате ссылки. Чтобы понять, почему, рассмотрим функцию operator+ для сложения двух матриц, где о левой матрице точно известно, что она является rvalue (а следовательно, может повторно использовать свою память для хранения суммы матриц):

```
1 Matrix operator+ ( matrix&& lhs , const Matrix& rhs ) // Возврат по значению
2 {
3     lhs += rhs ;
4     return std:: move(lhs); // Перемещение lhs в
5 } // возвращаемое значение
```

Подробнее про RVO