4.21 Rvalue-ссылки, их сходства и различия с обычными ссылками. Правила инициализации rvalue-ссылок. Примеры передачи параметров по rvalue-ссылке, перегрузка между обычными и rvalue-ссылками, логика выбора версии функции в случае такой перегрузки. Правила сворачивания ссылок (reference collapsing).

Неконстантная lvalue reference позволяет себя инициализировать только значениями lvalue, а **константные lvalue reference** значениями и lvalue и rvalue.

Rvalue reference позволяет себя инициализировать только rvalue-выражениями.

```
int x = 7;
int &lref = x; // инициализация ссылки l-value переменной x (значение l-value)
int &&rref = 7; // инициализация ссылки r-value литералом 7 (значение r-value)
```

Рассмотрим как работают rvalue references

```
int x = 0;
int& rx = x;
int& rx = 1; // WRONG! - надо присваивать lvalue
const int& crx = 1; // ок - это const ref и ей можно присваивать rvalue

int&& rrx = 1; // ок - присваиваем rvalue
int&& rrx = x; // WRONG! - надо присваивать rvalue

// Как создать rvalue из чего угодно? А вот так:
int&& ref1 = std::move(x); // This works fine

rrx = x; // ок - мы не инициализируем, а присваиваем значение из x
rrx = 5; // ок - аналогично

// при этом заметим:
x = 1; // rrx is still 5;
int& another_ref = rrx; // ок - rrx это идентификатор (т.е. lvalue)
int&& another_ref_2 = rrx; // WRONG! - потому что rrx не rvalue
```

В строчках 4 и 6 происходит продление жизни объекта, ссылка создается и будет жить на стеке пока не выйдет из области видимости. При этом переменная может сразу принять ислам, как если бы мы делали `String&& ref = String("aaaa")`. Строка ”aaaa” удаляется сразу после инициализации ссылки, а ref продолжает быть валидным и указывать на ”aaaa”.

Передача параметров по ссылкам

```
void fun(const int &lref) { // перегрузка функции для работы с l-values
  std::cout << "l-value reference to const\n";
}

void fun(int &&rref) { // перегрузка функции для работы с r-values
  std::cout << "r-value reference\n";
}

int main() {
  int x = 7;
  fun(x); // аргумент l-value вызывает функцию с ссылкой l-value
  fun(7); // аргумент r-value вызывает функцию с ссылкой r-value
}
```

Замечание: Если есть конструктор от const lvalue-ссылки и от rvalue-ссылки, компилятор отдаст rvalue объект во второй конструктор, так как **сработает перегрузка.** Хотя обе функции вроде бы подходят, второй случай считается perfect match

Правила сворачивания ссылок (reference collapsing):

- & + & = &
- & + && = &
- && + & = &
- && + && = &&

Свертывание ссылок происходит в контекстах, таких как

- инстанцирование шаблона
- определение auto переменных

Рассмотрим данный фрагмент кода:

```
template <typename T>
void func(T t) {
  T& k = t;
}

int main() {
    int i = 4;
    func<int&>(i);
}
```

При инстанцировании шаблона T установится равным int&. Какой же тип будет у переменной k внутри функции? Компилятор «увидит» int& & – а так как это запрещенная конструкция, компилятор просто преобразует это в обычную ссылку по правилу сворачивания ссылок.

4.22 Идея универсальных ссылок. Реализация функции std::move. Объяснение действия этой функции. Объяснение, почему принимаемый и возвращаемый типы именно такие.

Некоторые функции должны уметь принимать в качестве параметров как lvalue-reference, так и rvalue-reference. Здесь к нам на помощь приходят **universal references**. Универсальная ссылка обязана быть шаблонным параметром функции.

```
template <class T>
    void func(T&& t) {
}

func(4);              // 4 это rvalue: T становится int

double d = 3.14;
func(d);              // d это lvalue; T становится double&

float f() {...}
func(f());            // f() это rvalue; T становится float
```

Пояснение: Если была передана lvalue типа U, то T становится U& и `decltype(t) = int&` Если же U это rvalue, то T становится просто U и `decltype(t) = int&&`

Реализация std::move

```
template<typename T>
std::remove_reference_t<T>&& move(T&& param) {
    return static_cast<std::remove_reference_t<T>&&>(param);
}
```

Функция принимает универсальную ссылку так как задача move: превратить все (а мы не знаем что именно нам передали) в rvalue. Для возвращаемого типа используем type_traits, и поэтому делаем каст нашего типа к rvalue.

Замечание: `std::move` надо писать когда хотим из lvalue сделать rvalue, для rvalue объектов все и так будет работать правильно

4.23 Проблема прямой(идеальной) передачи. Предназначение функции emplace_back, ее преимущество перед push_back. Правильное использование функции std::forward (без реализации) для реализации механизма perfect forwarding.

Проблема: Когда у нас в шаблонную функцию передается пременное число аргументов, и мы не знаем какие из них rvalue, а какие lvalue, встает вопрос как передать дальше как rvalue те и только те, что изначально были rvalue?

Решение: Можем воспользоваться функцией std::forward

Пример **(механизм perfect forwarding)**:

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

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

Реализация std::forward

```
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.

push_back и emplace_back

Note: emplace_back есть во всех контейнерах, в котрорых есть push_back. Для контейнеров, в которых есть только insert, есть аналогичная функция emplace.

Решение проблемы с push_back в векторе (см билет 3.30) воплощено в функции emplace_back, которая семантически аналогична push_back. Однако в emplace_back не создается промежуточного временного объекта, так как в construct передаем сразу (args...), где args... - аргументы добавляемого объекта, которые **пробросятся сразу в конструктор**, и таким образом временная строка (которая потом бы скопировалась) не будет создана. То есть строка создается единственный раз и сразу на нужном месте.

```
template<typename... Args>
void emplace_back(const Args&... args){
    if(sz == cp) reserve(2 * cp);
    AllocTraits::construct(alloc, arr + sz, args...);
}
```

Преимущество emplace_back над push_back

```
std::vector<std::string> v;

v.push_back(std::string("abc")); // дважды создается строка
// первый - для передачи в функцию, второй - создается тот,
// который будет лежать в векторе (копируется в constuct)

v.emplace_back("abc"); // создается единожды
// сразу на нужном месте с нужным значением
```

На самом деле, emplace_back не решает проблему с излишним копированием, а только переносит ее на другой уровень. Ведь если среди аргументов, которые мы передаем в конструктор так же будут нетривиальные для копирования объекты, то в результате того, что мы передаем эти аргументы по констатнтой ссылке, будут вызваны лишние копирования.