5.4. Шаблонные шаблонные параметры, синтаксис использования. Пример: класс Stack на основе шаблонного контейнера. Шаблоны с переменным количеством аргументов (variadic templates). Синтаксис использования, пример: функция print. Пакеты аргументов и пакеты параметров, их распаковка. Реализация структуры is_homogeneous. Оператор “sizeof...”.

Шаблонные шаблонные параметры: В качестве параметров шаблона можно передавать другие шаблоны. Ниже приведен синтаксис использования на примере класса Stack.

```
template <typename T, template <typename U, typename Alloc> class Container
    = std::vector>
class Stack {
    Container<T, std::allocator<T>> c;
};

int main() {
    Stack<int, std::vector> s;
}
```

Variadic templates: Так же можно создавать шаблоны с переменным количеством параметров. Рассмотрим синтаксис на примере функции print от произвольного количества аргументов.

```
void print () {};

template <typename Head, typename... Tail> // Tail - пакет типов
void print(const Head& head, const Tail&... tail) { // tail - пакет аргументов
    std::cout << head << ' ';
    print(tail...); // многоточие распаковывает пакет (нужно для передачи в функцию)
}
```

Компилятор сгенерирует функции print от всех возможных наборов аргументов, от которых она вызывалась в программе. Возникает вопрос: зачем объявлять функцию print без аргументов? Почему нельзя было просто сделать так?

```
if (sizeof...(tail) == 0) { // оператор sizeof... возвращает
    print(tail...);         // количество элементов в пакете tail
}
```

Ответ прост: да, функция без аргументов в этом случае не вызовется, но она обязана будет скомпилироваться. Если не объявить ее получим CE.

Метафункция `is_homogeneous` - обобщение `is_same` на произвольное количество аргументов:

```
template <typename First, typename Second, typename... Types>
struct is_homogeneous {
    static const bool value = is_same_v<First, Second> &&
                              is_homogeneous<Second, Types...>::value;
};

template <typename First, typename Second>
struct is_homogeneous<First, Second> {
    static const bool value = is_same_v<First, Second>;
};
```

5.5. Зависимые имена. Пример неоднозначности между declaration'ом и expression'ом. Применение ключевого слова typename для устранения неоднозначности с зависимым именем. Применение слова template для решения аналогичной проблемы с зависимыми именами шаблонов (иллюстрация примером).

Рассмотрим такой пример

```
template <typename T>          template <>
struct S {                     struct S<int> {
    using X = T;                   static int X;
};                             };

template <typename T>
void f() {
    S<T>::X* a;
}
```

Заметим, что строчку из функции f можно трактовать двояко

1. declaration: объявляем переменную a типа `S<T>::X*`
2. expression: умножаем переменную `S<T>::X` на переменную a

В данном случае X является зависимым именем, то есть его смысл зависит от шаблонного параметра T (может даже не зависеть в данный момент, но потенциально зависеть). По умолчанию компилятор читает такие случаи как expression. Чтобы исправить это, необходимо написать `typename S<T>::X* a`. Теперь эта строчка будет читаться как объявление переменной a.

Замечание: Неважно какой знак там стоит (хоть никакого), важно что компилятор пытается трактовать все зависимые имена как переменные и не может скомпилировать эту строку без слова typename.

Рассмотрим другой пример

```
template <typename T>              template <>
struct SS {                        struct SS<int> {
    template <int M, int N>            static const int A = 0;
    struct A {};                   };
};

template <typename T>
void g() {
    SS<T>::A<1, 2> a;
}
```

И снова получаем неоднозначность

1. Объявляем переменную a типа `S<T>::A<1,2>`
2. `(SS<T>::A < 1), (2 > a)`

Одного слова typename нам тут уже не хватит, так как когда компилятор видит слово typename он ждет название типа, а в данном случае он получает название шаблона. Эта проблема решена с помощью слова template: `typename<T>::template A<1, 2> a;`

Замечание: Иногда нам может понадобиться слово template, но не понадобиться typename, например при использовании шаблонных функций и т. д. (то есть не классов)