TUPLE FOR EACH

Task

Вызывать func для каждого элемента кортежа

```
std::tuple<char, int, std::string> t{'a', 1, "abc"};
auto func =
    [](const auto& arg) { std::cout << arg << " "; };
```

Решение в лоб

```
int main() {
    std::tuple<char, int, std::string> t{'a', 1, "abc"};
    auto func =  [](const auto& arg) {
        std::cout << arg << " ";
    };

    func(std::get<0>(t));
    func(std::get<1>(t));
    func(std::get<2>(t));
}
```

Вынесем вызовы `func` в отдельную фукнцию

Решение для 3-кортежей

```
template <class Tuple, class Func>
void for_each(const Tuple& tuple, Func&& func) {
    func(std::get<0>(tuple));
    func(std::get<1>(tuple));
    func(std::get<2>(tuple));
}

int main() {
   std::tuple<char, int, std::string> t{'a', 1, "abc"};
   auto func =  [](const auto& arg) {
       std::cout << arg << " ";
   };
   for_each(t, func);
}
```

Избавимся от дублирования, используя variadic templates

Решение для 3-кортежей

```
template <size_t... Is, class Tuple, class Func>
void for_each_impl(const Tuple& tuple, const Func& func)
    (func(std::get<Is>(tuple)), ...);
}

template <class Tuple, class Func>
void for_each(const Tuple& tuple, const Func& func) {
    for_each_impl<0,1,2>(tuple, func);
}

int main() {
   std::tuple<char, int, std::string> t{'a', 1, "abc"};
   auto func = [](const auto& arg) {
       std::cout << arg << " ";
   };
   for_each(t, func);
}
```

Посмотрим на инстанциации шаблонов на cppinsights: Click me

Решение для 3-кортежей

```
template <size_t... Is, class Tuple, class Func>
void for_each_impl(const Tuple& tuple, const Func& func)
    (func(std::get<Is>(tuple)), ...);
}

template <class Tuple, class Func>
void for_each(const Tuple& tuple, const Func& func) {
    for_each_impl<0,1,2>(tuple, func);
}
```

Сейчас приходится вручную задавать индексы, которые попадают в пакет параметров `Is`.

Поэтому это решение только для 3-кортежей. Если научимся формировать `Is` в зависимости от размера кортежа, то получим общее решение.

Упаковка индексов в пакет параметров

Если в функцию передать нечто такое, что зависит от `Is`, то `Is` могут быть выведены. Пример:

```
template <size_t... Is>
struct IndexSequence {};

template <size_t... Is>
void f(IndexSequence<Is...>) {};

int main() {
    f(IndexSequence<0,1,2>{});
    f(IndexSequence<0,1,2,3,4>{});
}
```

Смотрим на специализации шаблона `f`: Click me

Решение для 3-кортежей

Используем этот подход в своем решении:

```
template <size_t... Is>
struct IndexSequence {};

template <size_t... Is, class T, class F>
void for_each_impl(const T& t, const F& f,
                   IndexSequence<Is...>) {
    (f(std::get<Is>(t)), ...);
}

template <class T, class F>
void for_each(const T& t, const F& f) {
    for_each_impl(t, f, IndexSequence<0,1,2>{});
}
```

Имена параметров сокращены для компактности кода

Строим IndexSequence для кортежа

Используя helper-метафункцию tuple_size мы можем получить размер кортежа по его типу в compile time:

```
using TupleType = std::tuple<char, int, std::string>;
static_assert(std::tuple_size<TupleType>::value == 3);
```

Если научимся для заданного числа `N` строить `IndexSequence<0,1,2,…,N-1>`, то задача будет решена

Строим IndexSequence для N

Напишем метафункцию `MakeIndexSequence`, которая работает следующим образом:

```
static_assert(std::is_same_v<MakeIndexSequence<>::type, IndexSequence<>>);
static_assert(std::is_same_v<MakeIndexSequence<1>::type, IndexSequence<0>>);
static_assert(std::is_same_v<MakeIndexSequence<2>::type, IndexSequence<0, 1>>);
static_assert(std::is_same_v<MakeIndexSequence<3>::type, IndexSequence<0, 1, 2>>);
```

Строим IndexSequence для N

```
template <size_t N>
struct MakeIndexSequence {
      using type = typename append_helper<
                        typename MakeIndexSequence<N - 1>::type,
                        N - 1
                   >::type;
};

template <>
struct MakeIndexSequence<0> {
    using type = IndexSequence<>;
};
```

Метафункция append_helper принимает IndexSequence и индекс, и добавляет этот индекс в конец IndexSequence:

```
using ActualType = append_helper<
                        IndexSequence<0>,
                        1
                   >::type;
using ExpectedType = IndexSequence<0, 1>;
static_assert(std::is_same_v<ActualType, ExpectedType>);
```

Пишем append_helper...

Строим IndexSequence для N

```
template <class IndexSequence, size_t Idx>
struct append_helper;

template <size_t LastIdx, size_t... Previous>
struct append_helper<IndexSequence<Previous...>, LastIdx> {
    using type = IndexSequence<Previous..., LastIdx>;
};

template <size_t N>
struct MakeIndexSequence {
      using type = typename append_helper<
                        typename MakeIndexSequence<N - 1>::type,
                        N - 1
                   >::type;
};

template <>
struct MakeIndexSequence<0> {
    using type = IndexSequence<>;
};
```

Все готово! Можем использовать `tuple_size` и `MakeIndexSequence` , чтобы дописать фукнкцию `for_each`

Итоговое решение

```
template <size_t... Is, class T, class F>
void for_each_impl(const T& t, const F& f,
                   IndexSequence<Is...>) {
    (f(std::get<Is>(t)), ...);
}

template <class T, class F>
void for_each(const T& t, const F& f) {
    using index_sequence =
        typename MakeIndexSequence<std::tuple_size_v<T>>
    for_each_impl(t, f, index_sequence{});
}
```

Замечание I

Самописные `MakeIndexSequence` и `IndexSequence` можно заменить аналогами из стандартной библиотеки: `std::index_sequence` и `std::make_index_sequence`

```cpp
template <size_t... Is, class T, class F>
void for_each_impl(const T& t, const F& f, std::index_sequence<Is...>) {
    (f(std::get<Is>(t)), ...);
}

template <class T, class F>
void for_each(const T& t, const F& f) {
    using index_sequence = std::make_index_sequence<std::tuple_size_v<T>>;
    for_each_impl(t, f, index_sequence{});
}
```

Замечание II

Оператор "запятая" может быть перегружен, поэтому могут быть выполнены какие-то дополнительные вычисления в `(f(std::get<Is>(t)), …);`

```
struct T {};
T operator,(const T&, const T&) {
    std::cout << 1; return T();
}

int main() {
    std::tuple<T, T, T> t;
    for_each(t, [](const auto& x) { return x; });
}
```

Замечание II

Перегрузка оператора "запятая" не будет использоваться, если один из операндов имеет тип void. Можно добиться этого так:

```
template <size_t... Is, class T, class F>
void for_each_impl(const T& t, const F& f, std::index_sequence<Is...>) {
    ((void)f(std::get<Is>(t)), ...);
}
```

либо так:

```
template <size_t... Is, class T, class F>
void for_each_impl(const T& t, const F& f, std::index_sequence<Is...>) {
    ((f(std::get<Is>(t)), (void)0), ...);
}
```

Замечание III

Что делать, если стандарт языка не поддерживает fold-expressions (since c++11, until c++17)?

```cpp
template <size_t... Is, class T, class F>
void for_each_impl(const T& t, const F& f, IndexSequence<Is...>) {
    std::initializer_list<int> lst = {(f(std::get<Is>(t)), 0)...};
    (void)lst; // suppress warning
}

template <class T, class F>
void for_each(const T& t, const F& f) {
    using index_sequence = typename MakeIndexSequence<std::tuple_size<T>::value>::type;
    for_each_impl(t, f, index_sequence{});
}
```

Создаем std::initializer_list, используя паттерн
`( (void)f(std::get<Is>(t)), 0 )`

`std::index_sequence` появился только в c++14, поэтому используем свою реализацию.

Замечание III

Почему именно такой паттерн?

- initializer_list хранит объекты одного типа, поэтому выражение в паттерне должно иметь всегда один и тот же тип. Это достигается за счет использования оператора "запятая" с нулем в качестве второго операнда.
- приведение результата функции к `void` нужно, чтобы избежать использования перегрузки оператора "запятая"