КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 23 — Кортежи

Преподаватель: Поденок Леонид Петрович, 505а-5

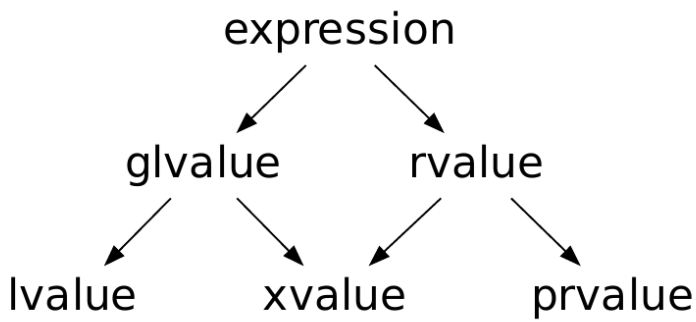
+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Категории значений



lvalue может появляться в левой части выражения присваивания и обозначает функцию или объект.

xvalue (значение «e**X**piring») относится к объекту, обычно ближе к концу его времени жизни, например, его ресурсы могут быть перемещены. ***xvalue*** — это результат некоторых видов выражений, включающих ссылки на ***rvalue***. Например, результатом вызова функции, возвращаемый тип которой является ссылкой на ***rvalue***, является значение ***xvalue***.

glvalue («обобщенное» *lvalue*) – это ***lvalue*** или ***xvalue***. ***glvalue***-выражение – это выражение, вычисление которого определяет индивидуальность объекта, битового поля или функции.

rvalue – представляет собой ***xvalue***, временный объект или подобъект или значение, которое вообще не связано с объектом (литерал).

prvalue («чистое» ***rvalue***) – это ***rvalue***, которое не является ***xvalue***. Например, результатом вызова функции, возвращаемый тип которой не является ссылкой, является *prvalue*-значение. Значение литерала, такого как **`12`**, **`7.3e5`** или **`true`**, также является prvalue-значением.

Например, встроенные операторы присваивания ожидают, что левый операнд является **`lvalue`**, а правый операнд является **`prvalue`** и в результате этот оператор выдает **`lvalue`**.

Спецификатор типа decltype

`decltype ( `*выражение*` )`

Тип, обозначенный **`decltype(e)`**, определяется следующим образом:

- если **e** — это id-выражение без скобок или доступ к члену класса без скобок (с помощью **`.`** или **`->`**), то **`decltype(e)`** — это тип той сущности, которая называется **e** (если такой сущности нет или если **e** называет перегруженные функции, программа считается сформированной неправильно);
- в противном случае, если **e** — xvalue, то **`decltype(e)`** — это **`T&&`**, где **T** - это тип **e**;
- в противном случае. если

otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e;

- в противном случае, если **e** — это **`lvalue`**, то **`decltype(e)`** - это **`T&`**, где **T** — это тип **e**;
- в противном случае **`decltype(e)`** — это просто тип **e**.

Пример

```
const int&& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1 = i; // type is const int&&
decltype(i) x2; // type is int
decltype(a->x) x3; // type is double
decltype((a->x)) x4 = x3; // type is const double&
```

std::move — переместить как rvalue

```
#include <utility>

template <class T>
typename remove_reference<T>::type&& move(T&& arg) noexcept;
```

Выполняет безусловное приведение передаваемого **lvalue**-аргумента в **rvalue**-ссылку.

Это вспомогательная функция для принудительного применения семантики перемещения значений, даже если у них есть имя. При этом непосредственное использование возвращаемого значения приводит к тому, что **arg** считается **rvalue**.

Как правило, **rvalue** — это значения, адрес которых не может быть получен путем разыменования, либо потому, что они являются литералами, либо потому, что они временны по своей природе, как, например, значения, возвращаемые функциями или явными вызовами конструктора.

Путем передачи объекта в эту функцию получается относящееся к этому объекту **rvalue**.

Многие компоненты стандартной библиотеки реализуют семантику перемещения, позволяя передавать владение активами и свойствами объекта напрямую без необходимости их копирования, если аргумент представляет собой **rvalue**.

В стандартной библиотеке перемещение подразумевает, что перемещенный объект остается в допустимом, но неизвестном состоянии. Это означает, что после такой операции значение перемещенного объекта должно быть уничтожено или ему должно быть присвоено новое значение. В противном случае доступ к нему дает неопределенное значение.

std::forward — «пропускает» через себя аргумент

```
#include <utility>

template <class T> T&& forward(typename remove_reference<T>::type& arg) noexcept;
template <class T> T&& forward(typename remove_reference<T>::type&& arg) noexcept;
```

Функция возвращает **arg** без изменения его типа.

Аргументы **lvalue** передается, как **lvalue**, аргументы не **lvalue** – как **rvalue**.

Это вспомогательная функция позволяет создавать функциии-обертки, которые идеально передают аргументы без изменения их типа, сохраняя при этом любую возможную семантику перемещения.

Необходимость в этой функции проистекает из того факта, что все именованные значения (например, параметры функции) всегда вычисляются как **lvalue**, даже те, которые объявлены как **rvalue**-ссылки, и это создает потенциальные трудности в сохранении семантики перемещения в шаблонных функциях, которые пересылают аргументы другим функциям.

Обе сигнатуры возвращают

```
static_cast<decltype(arg)&&>(arg)
```

Надо отметить, что при любом создании экземпляра требуется явно указывать тип **T** (неявный вывод для **T** не работает).

Пример forward()

```
#include <utility>      // std::forward
#include <iostream>     // std::cout

// перегрузки функций со ссылками lvalue и rvalue:
void overloaded(const int& x) {std::cout << "[lvalue]";}
void overloaded(int&& x)      {std::cout << "[rvalue]";}

// шаблон функции, принимающий rvalue-ссылку чтобы вывести тип:
template <class T> void fn(T&& x) {
  overloaded(x);                    // всегда lvalue
  overloaded(std::forward<T>(x));  // rvalue, если аргумент -- rvalue
}

int main () {

  int a;

  std::cout << "Вызов fn с lvalue: ";
  fn(a);
  std::cout << '\n';

  std::cout << " Вызов fn с rvalue: ";
  fn(0);
  std::cout << '\n';
}
```

Вывод

```
Вызов fn с lvalue: [lvalue][lvalue]
Вызов fn с rvalue: [lvalue][rvalue]
```

std::make_pair — создать объект типа pair

```
#unclude <utility>

template <class T1, class T2>
pair<V1, V2> make_pair(T1&& x, T2&& y); // что-то типа emplace
```

Создает объект с типом **pair**, для первого элемента которого задается значение **x**, а для второго — значение **y**.

Типы шаблонов могут быть неявно выведены из аргументов, переданных функции **make_pair()**.

Объекты типа **pair** могут быть созданы из объектов других типов **pair**, если соответствующие типы элементов неявно конвертируемы.

Функция возвращает

```
pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y))
```

Если **T1** и/или **T2** являются ссылками на **rvalue**, объекты перемещаются, а **x** и/или **y** остаются в неопределенном, но допустимом состоянии.

Пары значений

```
#unclude <utility>

template <class T1, class T2> struct pair;
```

Этот класс объединяет пару значений, которые могут быть разных типов (**T1** и **T2**). Доступ к отдельным значениям можно получить через его публичные члены **first** и **second**.

Пары — это частный случай кортежа.

Переменные-члены

Член	Определение
first	Первое значение в паре
second	Второе значение в паре

Типы членов

Тип члена	Определение	Замечания
first_type	Первый параметр шаблона (**T1**)	Тип члена first
second_type	Второй параметр шаблона (**T2**)	Тип члена second

Открытые функции-члены

(**`constructor`**) — конструктор пары
`pair::operator=` — присвоить содержимое
`pair::swap` — обменять содержимое

Перегруженные функции не члены

`relational operators (pair)` — реляционные операторы
`swap (pair)` — обменять содержимое двух пар
`get (pair)` — получить элемент

Специализации классов

`tuple_element<pair>` — тип элемента кортежа для пары
`tuple_size<pair>` — характеристики кортежа для пары

Конструкторы

Включает в себя индивидуальное создание двух компонентных объектов пары с инициализацией, которая зависит от вызываемой формы конструктора.

(1) По умолчанию

```
constexpr pair();
```

Создает объект с инициализированными значениями элементов.

(2) Копирования/перемещения с неявным преобразованием

```
template <class U, class V> pair(const pair<U, V>& pr);
template <class U, class V> pair(pair<U, V>&& pr);
                            pair(const pair& pr) = default;
                            pair(pair&& pr)      = default;
```

Объект инициализируется содержимым объекта пары **pr**.
Конструктору каждого из его членов передается соответствующий член пары **pr**.

pr — другой объект типа **pair**. Это может быть объект того же типа, что и конструируемый объект, или тип пары, типы элементов которой неявно могут быть преобразованы в типы создаваемой пары.

(3) С раздельной инициализацией

```
                              pair(const first_type& a, const second_type& b);
template<class U, class V> pair(U&& a, V&& b);
```

Первый элемент создается с помощью **a**, а второй — с помощью **b**.

a, **b** — объекты типа **first**, **second**, соответственно, или некоторого другого типа, который может быть неявно сконвертирован в них.

(4) Поэлементный

```
template <class... Args1, class... Args2>
  pair(piecewise_construct_t pwc,
       tuple<Args1...> first_args,
       tuple<Args2...> second_args);
```

Создает элементы **first** и **second** на месте, передавая элементы **first_args** в качестве аргументов конструктору **first** и элементы **second_args** конструктору **second**.

Пример

```
#include <utility>      // std::pair, std::make_pair
#include <string>       // std::string
#include <iostream>     // std::cout

int main () {

  std::pair <std::string, double> product1;                     // default c-tor
  std::pair <std::string, double> product2("tomatoes", 2.30); // value init
  std::pair <std::string, double> product3(product2);          // copy  c-tor
  // с использованием функции make_pair(move)
  product1 = std::make_pair(std::string("lightbulbs"),0.99);  //
  product2.first  = " shoes";                                  // Тип first  -- string
  product2.second = 39.90;                                     // Тип second -- double

  std::cout << "Цена " << product1.first << " - " << product1.second << "p.\n";
  std::cout << "Цена " << product2.first << " - " << product2.second << "p.\n";
  std::cout << "Цена " << product3.first << " - " << product3.second << "p.\n";
}
```

Вывод

```
Цена lightbulbs - 0.99p.
Цена shoes - 39.9p.
Цена tomatoes -2.3p.
```

pair::operator= — присвоить содержимое

(1) Присваивание копированием

```
                              pair& operator= (const pair& pr);
template <class U, class V> pair& operator= (const pair<U, V>& pr);
```

(2) Присваивание перемещением

```
pair& operator= (pair&& pr)
           noexcept(is_nothrow_move_assignable<first_type>::value &&
                    is_nothrow_move_assignable<second_type>::value);
template <class U, class V> pair& operator= (pair<U, V>&& pr);
```

Присваивает значение pr в качестве нового содержимого объекта типа **pair**.

Члену **first** присваивается значение **pr.first**, члену **second** — значение **pr.second**.

Копирующие формы (1) выполняют копирование, при этом члены аргумента сохраняют свои значения после вызова.

Формы перемещения (2) выполняют присвоение перемещением (как если бы вызывалась **forward( )** для каждого члена), что для членов с типом, поддерживающим семантику перемещения, подразумевает, что эти члены **pr** остаются в неопределенном, но допустимом состоянии.

Пример

```
// pair::operator= example
#include <utility>      // std::pair, std::make_pair
#include <string>       // std::string
#include <iostream>     // std::cout

int main () {

    std::pair <std::string, int> planet;
    std::pair <std::string, int> homeplanet;

    planet = std::make_pair("Земля", 6371);

    homeplanet = planet;

    std::cout << "  Наша планета: " << homeplanet.first  << '\n';
    std::cout << "Радиус планеты: " << homeplanet.second << "км\n"';
    return 0;
}
```

Вывод

```
  Наша планета: Земля
Радиус планеты: 6371 км
```

Кортежи

Кортежи — это объекты, которые объединяют элементы, возможно, разных типов в один объект, точно так же, как парные объекты делают для пар элементов, но обобщены для любого количества элементов.

Концептуально они похожи на простые старые структуры данных (структуры, подобные C), но вместо именованных элементов данных доступ к их элементам осуществляется по их порядку в кортеже.

Выбор конкретных элементов в кортеже выполняется на уровне создания экземпляра шаблона, и, следовательно, он должен быть указан во время компиляции с помощью вспомогательных функций, таких как **`get`** и **`tie`**.

Класс кортежа тесно связан с классом пары (**`pair`** из **`<utility>`**) — кортежи могут быть построены из пар, а пары для определенных целей могут обрабатываться как кортежи.

Шаблоны классов

`tuple`	— кортеж
`tuple_size`	— характеристики размера кортежа
`tuple_element`	— тип элемента кортежа

Шаблоны функций создания объектов и доступа к ним

`make_tuple`	— создать кортеж
`forward_as_tuple`	— передать как кортеж
`tie`	— привязать аргументы к элементам кортежа
`tuple_cat`	— соединить кортежи
`get`	— получить элемент

tuple — кортеж

```
template <class... Types> class tuple;
```

Кортеж — это объект, способный хранить коллекцию элементов.
Каждый элемент может быть разного типа.

Шаблоны открытых функций-членов

`constructor` — создание кортежа
`tuple::operator=` — присвоить содержимое
`tuple::swap` — обменять содержимое

Шаблоны открытых функций не членов

`relational operators(tuple)` — реляционные операторы
`swap(tuple)` — обменять содержимое двух кортежей
`get(tuple)` — получить элемент

std::tuple::tuple — создание кортежа

Создание кортежа включает в себя индивидуальное построение его элементов с их инициализацией, которая зависит от вызываемой формы конструктора.

(1) По умолчанию

```
constexpr tuple();
```

Создает объект кортежа с инициализированными значениями элементов.

(2) Копирования и перемещения

```
tuple(const tuple& tpl) = default;
tuple(tuple&&      tpl) = default;
```

Объект инициализируется содержимым объекта кортежа **tpl**.

Соответствующий элемент **tpl** передается конструктору каждого элемента.

tpl — другой объект кортежа с таким же количеством элементов.

Это может быть объект того же типа, что и конструируемый объект (2 — к-тор пермещения), или тип кортежа, типы элементов которого неявно преобразуются в типы конструируемого объекта кортежа (1 — к-тор копирования).

Большинство форм имеют две сигнатуры — одна принимает ссылки на **const lvalue**, которые копируют элементы в кортеж, а другая принимает ссылки **rvalue**, которые их вместо этого перемещают, если их типы поддерживают семантику перемещения.

(3) Неявного преобразования

```
template <class... Utypes> tuple(const tuple<UTypes...>& tpl);
template <class... Utypes> tuple(tuple<UTypes...>&&      tpl);
```

То же, что и выше, при этом все типы в **tpl** должны быть неявно преобразованы в тип соответствующего элемента в созданном объекте кортежа.

(4) С инициализацией

```
                          explicit tuple (const Types&... elems);
template <class... Utypes> explicit tuple (UTypes&&...    elems);
```

Инициализирует каждый элемент соответствующим элементом из **elems**.

Соответствующий элемент **elems** передается конструктору каждого элемента.

elems ... — ряд элементов, каждый из которых соответствует типу в параметрах шаблона **Types** или **UTypes**.

(5) Преобразование из пары

```
template <class U1, class U2> tuple(const pair<U1,U2>&  pr);
template <class U1, class U2> tuple(      pair<U1,U2>&& pr);
```

В объекте есть два элемента, соответствующие **pr.first** и **pr.second**. Все типы в **pr** должны быть неявно преобразованы в тип соответствующего элемента в объекте кортежа.

pr — объект пары, первый и второй типы которого (**U1** и **U2**) соответствуют тем, которые используются в качестве аргументов шаблона класса, или могут быть неявно преобразованы в них.

(6) Со своим аллокатором

```
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, const tuple& tpl);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, tuple&& tpl);
template<class Alloc,class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, const tuple<UTypes...>& tpl);
template<class Alloc, class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, tuple<UTypes...>&& tpl);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, const Types&... elems);
template<class Alloc, class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, UTypes&&... elems);
template<class Alloc, class U1, class U2>
  tuple (allocator_arg_t aa, const Alloc& alloc, const pair<U1,U2>& pr);
template<class Alloc, class U1, class U2>
  tuple (allocator_arg_t aa, const Alloc& alloc, pair<U1,U2>&& pr);
```

То же, что и все версии выше, за исключением того, что каждый элемент создается с использованием аллолкатора **alloc**.

aa — значение **std::allocator_arg**. Это постоянное значение используется просто для выбора форм конструктора с параметром аллокатора.

alloc — объект-аллокатор, используемый для создания элементов.

Пример создания кортежей

```
#include <iostream>     // std::cout
#include <utility>      // std::make_pair
#include <tuple>        // std::tuple, std::make_tuple, std::get

int main () {

  std::tuple<int, char>  first;                              // default
  std::tuple<int, char>  second(first);                      // copy
  std::tuple<int, char>  third(std::make_tuple(20,'b')); // move
  std::tuple<long, char> fourth(third);                      // implicit conversion
  std::tuple<int, char>  fifth(10,'a');                      // initialization
  std::tuple<int, char>  sixth (std::make_pair(30,'c')); // from pair/move

  std::cout << "sixth contains: " << std::get<0>(sixth);
  std::cout << " and " << std::get<1>(sixth) << '\n';

  return 0;
}
```

Вывод

```
sixth contains: 30 and c
```

std::tuple::operator= — присвоить содержимое

(1) Копирования/перемещения

```
tuple& operator= (const tuple&  tpl);
tuple& operator= (      tuple&& tpl) noexcept;
```

tpl — другой объект кортежа с таким же количеством элементов. Это может быть объект того же типа, что и конструируемый объект (2), или тип кортежа, типы элементов которого неявно преобразуются в типы конструируемого объекта кортежа.

(2) С неявным преобразованием

```
template <class... Utypes> tuple& operator= (const tuple<UTypes...>&  tpl);
template <class... Utypes> tuple& operator= (      tuple<UTypes...>&& tpl);
```

(3) С преобразованием из пары

```
template <class U1, class U2> tuple& operator= (const pair<U1,U2>&  pr);
template <class U1, class U2> tuple& operator= (      pair<U1,U2>&& pr);
```

Присваивает **tpl** (или **pr**) в качестве нового содержимого для объекта кортежа. Каждому из элементов в объекте кортежа присваивается соответствующий ему элемент **tpl** (или **pr**).

pr — объект пары, первый и второй типы которого (U1 и U2) соответствуют тем, которые используются в качестве аргументов шаблона класса, или могут быть неявно преобразованы в них.

Формы, принимающие ссылку **lvalue** в качестве аргумента, выполняют копирование, при этом элементы аргумента сохраняют свои значения после вызова. Формы, принимающие ссылку на **rvalue**, выполняют присваивание перемещения (как будто вызывая **forward( )** для каждого элемента), что для элементов типов, поддерживающих семантику перемещения, подразумевает, что эти элементы **tpl** (или **pr**) остаются в неопределенном, но допустимом состоянии.

Для каждого конструктора кортежа генерируется исключение только в том случае, если исключение вызывает конструктор одного из типов в **Types** .

Пример присваивания кортежей

```
#include <iostream>     // std::cout
#include <utility>      // std::pair
#include <tuple>        // std::tuple, std::make_tuple, std::get

int main () {

  std::pair<int, char> mypair (0, ' ');
  std::tuple<int, char>  a(10, 'x');
  std::tuple<long, char> b, c;

  b = a;                                  // copy assignment
  c = std::make_tuple(100L, 'Y');         // move assignment
  a = c;                                  // conversion assignment
  c = std::make_tuple(100, 'z');          // conversion / move assignment
  a = mypair;                             // from pair assignment
  a = std::make_pair(2, 'b');             // form pair /move assignment

  std::cout << "c contains: " << std::get<0>(c);
  std::cout << " and "        << std::get<1>(c) << '\n';

  return 0;
}
```

Вывод

```
c contains: 100 and z
```

std::get — получить элемент

Возвращает ссылку на **K**-й элемент кортежа **tpl**.

(1) **Принимает и возвращает ссылку на lvalue**

```
template <size_t K, class... Types>
typename tuple_element< K, tuple<Types...>>::type&
get(tuple<Types...>& tpl) noexcept;
```

tpl — объект кортежа с более чем **K** элементами.

K — позиция элемента в кортеже, где 0 является позицией первого элемента.

size_t — целочисленный тип без знака.

Types — типы элементов в кортеже (обычно получаются неявно из **tpl**).

(2) Принимает и возвращает ссылку на rvalue

```
template <size_t K, class... Types>
typename tuple_element< K, tuple<Types...>>::type&&
get(tuple<Types...>&& tpl) noexcept;
```

Версия (2) принимает в качестве аргумента ссылку на **rvalue** кортежа, а также применяет **forward()** к возвращаемому элементу.

(3)Константный кортеж и константный возврат

```
template <size_t K, class... Types>
typename tuple_element< K, tuple<Types...> >::type const&
get(const tuple<Types...>& tpl) noexcept;
```

Версия (3) принимает в качестве аргумента константный кортеж и возвращает константную ссылку на элемент.

get также можно использовать с объектами, подобными кортежу, таким как пара и массив.

Возвращаемое значение

Ссылка на элемент в указанной позиции в кортеже.

tuple_element< I, tuple <Types ...>>::type — это тип I-го элемента в кортеже.

Важно!!!

Параметры шаблона должны быть **constexpr** (константные значения времени компиляции).

Пример получения элемента

```
#include <iostream>
#include <tuple>

int main () {

  std::tuple<int, char> mytuple(10, 'a');

  std::get<0>(mytuple) = 20;

  std::cout << "mytuple содержит: ";
  std::cout << std::get<0>(mytuple) << " и " << std::get<1>(mytuple);
  std::cout << std::endl;

  return 0;
}
```

Вывод

```
mytuple содержит: 20 и a
```

std::tie — привязать аргументы к элементам кортежа

C++11

```
template<class... Types>
            tuple<Types&...> tie(Types&... args) noexcept;
```

C++14

```
template<class... Types>
  constexpr tuple<Types&...> tie(Types&... args) noexcept;
```

Создает объект кортежа, элементы которого являются ссылками на аргументы в **args** в том же порядке. Это позволяет набору объектов действовать как кортеж, что особенно полезно для распаковки объектов кортежа.

args — список объектов (**lvalue**), которые нужно связать как элементы кортежа.

Для указания элементов кортежа, которые будут игнорироваться вместо привязки к определенному объекту, может использоваться специальная константа **ignore**.

Возвращаемое значение

Кортеж со ссылками на аргументы lvalue.

Пример упаковки и распаковки кортежа

```
#include <iostream>     // std::cout
#include <tuple>        // std::tuple, std::make_tuple, std::tie

int main () {

  int  myint;
  char mychar;

  std::tuple<int, float, char> mytuple;
  mytuple = std::make_tuple(10, 2.6, 'a');        // упаковка в кортеж

  std::tie(myint, std::ignore, mychar) = mytuple; // распаковка в переменные
  auto tpl2 = std::tie(myint, std::ignore, mychar) = mytuple;

  std::cout << "myint содержит: "  << myint << '\n';
  std::cout << "mychar содержит: " << mychar << '\n';
  std::cout << "tpl2: <" << std::get<0>(tupp) << ", "
//                         << std::get<1>(tupp) << ">" << '\n';
                           << std::get<2>(tupp) << ">" << '\n';
  return 0;
}
```

Вывод

```
myint содержит: 10
mychar содержит: a
tpl2: <10, a>
```

std::make_tuple — создать кортеж

```
template<class... Types> tuple<VTypes...> make_tuple(Types&&... args);
```

Создает объект кортежа соответствующего типа, содержащий элементы, указанные в **args**.

Тип возвращаемого объекта (**tuple<VTypes ...>**) выводится из **Types** — в **VTypes** используется преобразованный в **rvalue** с помощью **std::decay** эквивалент каждого типа из **Types**.

Функция вызывает конструктор инициализации кортежа, пересылая ему аргументы, используя **forward()**.

args — список элементов, которые должен содержать построенный кортеж.

Возвращаемое значение

Объект кортежа соответствующего типа для хранения всех аргументов.

std::decay — «ослабленный» тип **T** — это тот же тип, который получается в результате стандартных преобразований, которые происходят, когда выражение **lvalue** используется в качестве **rvalue** с удаленным cv-квалификатором и преобразованием в указатель (массив, функция).

Пример

```
#include <iostream>
#include <tuple>
#include <functional>

int main() {

  auto first = std::make_tuple(10, 'a');           // tuple < int, char >

  const int a = 0;
  int b[3];                                        // decayed types:
  auto second = std::make_tuple(a, b);             // tuple < int, int* >
  auto third = std::make_tuple(std::ref(a),"abc"); // tuple< const int&,
                                                   //        const char* >
  std::cout << "third contains: " << std::get<0>(third);
  std::cout << " and " << std::get<1>(third);
  std::cout << std::endl;

  return 0;
}
```