Лекция

TEMPLATES

C++ - язык множества парадигм

- процедурное программирование
- объектно-ориентированное программирование
- обобщенное программирование

Alias templates

using сильнее, чем вы думаете

```
1.  typedef    std::map<std::string, size_t>    words_counter_t;


2.  template<class type>
3.  using obj_counter = std::map<type, size_t>;

4.  obj_counter<int> digit_counter;


5.  using ivec = std::vector<int>;
```

Variadic templates

- *Шаблоном с переменным числом параметров* (**variadic template**) называется шаблон функции или класса, параметризуемый некоторым набором разнообразных типов (**parameter pack**).

- Оператор *эллипсис(…) - многоточие*

- Не так уж много функций

Variadic templates

шаблон может принять 0 или более типов в качестве аргументов

```
1.   template<typename... Args>
2.   struct SomeType;
```

Variadic templates

шаблон может принять 0 или более типов в качестве аргументов

```
1.  template<typename... Args>
2.  struct SomeType;
```

```
3.  template <int ... Numbers>
4.  struct A;
```

легально?

Variadic templates

количество параметров в паке можно получить используя оператор sizeof…

```
1.  template <typename... Args>
2.  void foo(Args... args) {
3.      std::cout << sizeof...(args) << std::endl;
4.  }

5.  template <typename... Args>
6.  struct A {
7.      static const size_t number = sizeof...(Args);
8.  };


9.  int main() {
10.     foo(1, 2.3);
11.     std::cout << A<bool, int, int, int, int, int>::number;
12. }
```

Variadic templates

количество параметров в паке можно получить используя оператор sizeof…

```
1.  template <typename... Args>
2.  void foo(Args... args) {
3.      std::cout << sizeof...(args) << std::endl;
4.  }

5.  template <typename... Args>
6.  struct A {
7.      static const size_t number = sizeof...(Args);
8.  };


9.  int main() {
10.     foo(1, 2.3);
11.     std::cout << A<bool, int, int, int, int, int>::number;
12. }
```

Не путать sizeof(Args)…!

Variadic templates

распаковка пака параметров шаблона (pack expansion)

```
1.  template<typename... Args> // Объявление
2.  void foo(Args... args);    // Использование

3.  foo(1, 2.3, "abcd");
4.  foo<int, double, const char*>(1, 2.3, "abcd");
```

Variadic templates

можно использовать в качестве аргумента вызова функции

```
1.  template<typename T>
2.  T bar(T t) {
3.      // ...
4.  }

5.  template<typename... Args>
6.  void foo(Args... args) {
7.      //...
8.  }

9.  template<typename... Args>
10. void foo2(Args... args) {
11.     foo(bar(args)...);
12. }
```

Variadic templates

применять к нему операции каста и т.п.

```
1.  (const args&...)
2.  ((f(args) + g(args))...)
3.  (f(args...) + g(args...))
4.  (std::make_tuple(std::forward<Args>(args)...))
```

Variadic templates

применять к нему операции каста и т.п.

```
1.   (const args&...)          // -> (const T1& arg1, const T2& arg2, ...)
2.   ((f(args) + g(args))...)  // -> (f(arg1) + g(arg1), f(arg2) + g(arg2), ...)
3.   (f(args...) + g(args...)) // -> (f(arg1, arg2,...) + g(arg1, arg2, ...))
4.   (std::make_tuple(std::forward<Args>(args)...))
// -> (std::make_tuple(std::forward<T1>(arg1), std::forward<T2>(arg2), ...))
```

Variadic templates

пример

```
1.  template <typename... BaseClasses>
2.  class ClassName : public BaseClasses... {
3.  public:
4.      ClassName(BaseClasses&&... base_classes) :
5.          BaseClasses(base_classes)...
6.      {}
7.  };
```

Variadic templates

распаковка двух пачек одновременно

```
1.  // их размер должен совпадать!
2.  template <typename... Ts>
3.  struct Victim {
4.      template<typename... Us>
5.      static void fun() {
6.          std::initializer_list<Base*> args = { (new Derived<Ts, Us>())... };
7.      }
8.  };
```

распаковка пачки происходит изнутри - наружу

```
9.  // т.е. сначала раскрываются внутренние пачки, а затем внешние
10. template <typename... Ts, typename... Us>
11. using MMultipleElementsTuple = std::tuple<std::tuple<Ts..., Us>...>;
```

Variadic templates

пример

```
1.  template <typename T, typename... Ts>
2.  size_t hash_combine(const T& t, const Ts&... ts) {
3.      size_t seed = std::hash<T>()(t);
4.      if (sizeof...(ts) == 0) {
5.          return seed;
6.      }
7.      size_t remainder = hash_combine(ts...);  // no recursion!
8.      return hash_128_to_64(seed, remainder);
9.  }

10. struct Obj {
11.     int x;
12.     string y;
13.     float z;

14.     size_t hash() const { return hash_combine(x, y, z); }
15. }
```

Variadic templates

пример

```
1.  vector<Obj> v;

2.  v.emplace_back(param1, param2);
    // doesn't take Obj, takes parameters to MAKE an Obj

    // No copies, no moves.



3.  auto shared_obj = std::make_shared<Obj>(param1, param2);

4.  auto uniq_obj = std::make_unique<Obj>(param1, params2);
```

Variadic templates

особенности

- отсутствует возможность сохранить пачку типов шаблона для последующей её обработки
- нельзя использовать часть пачки
- нельзя получить элемент из пачки по индексу

Safe pointer to one of several types

```
1.  /*  Safe pointer to one of several types. */
2.  template <typename... Types>
3.  class DiscriminatedPtr
4.  {
5.      // <, not <=, as our indexes are 1-based (0 means "empty")
6.      static_assert(sizeof...(Types) < std::numeric_limits<uint16_t>::max(),
                      "too many types");
7.      // it relies on the fact that(on x86_64) there are 16 unused bits in a pointer
8.      uintptr_t data_; // pointer = the least significant 48 bits,
9.
10.     template <typename T>
11.     uint16_t typeIndex() const { // the most significant 16 bits of data_
12.         // 0 = empty ptr, or the 1-based type index of T in Types
13.         return uint16_t(dptr_detail::GetTypeIndex<T, Types...>::value);
14.     }
15.     uint16_t index() const { return data_ >> 48; }

16.     void* ptr() const { return reinterpret_cast<void*>(data_ & ((1ULL << 48) - 1)); }
17.
18.     void set(void* p, uint16_t v) {
19.         uintptr_t ip = reinterpret_cast<uintptr_t>(p);
20.         CHECK(!(ip >> 48));
21.         ip |= static_cast<uintptr_t>(v) << 48;
22.         data_ = ip;
23.     }
```

Safe pointer to one of several types

```
24. public:
25.     DiscriminatedPtr() : data_(0) {}
26.     template <typename T>
27.     explicit DiscriminatedPtr(T* ptr) { set(ptr, typeIndex<T>()); }

28.     bool empty() const { return index() == 0; } // true iff it is empty
29.     void clear() { data_ = 0; } // make it empty

30.     template <typename T> // set to point to an object of type T
31.     void set(T* ptr) { set(ptr, typeIndex<T>()); }
32.     template <typename T> // assignment operator from a pointer of type T
33.     DiscriminatedPtr& operator=(T* ptr) { set(ptr); return *this; }

34.     template <typename T> // true iff type T pointed
35.     bool hasType() const { return index() == typeIndex<T>(); }

36.     template <typename T> // returns pointer to, if it is of type T
37.     /*const*/ T* get_nothrow() /*const*/ noexcept {
38.         void* p = (hasType<T>()) ? ptr() : nullptr; // returns nullptr
39.         return static_cast<T*>(p); // if it is empty or points to a different type
40.     }
41.     template <typename T>
42.     /*const*/ T* get() /*const*/ {
43.         if (!hasType<T>()) throw std::invalid_argument("Invalid type");
44.         return static_cast<T*>(ptr());
45.     }
```

Safe pointer to one of several types

```
46.    template <typename V>
47.    typename dptr_detail::VisitorResult<V,Types...>::type apply(V&& visitor) /*const*/ {
48.       size_t n = index();
49.       if (n == 0)
50.           throw std::invalid_argument("Empty DiscriminatedPtr");
51.       return dptr_detail::ApplyVisitor<V,Types...>{}(n,std::forward<V>(visitor),ptr());
52.    }

53. }; // end of class DiscriminatedPtr


54. template <typename Visitor, typename... Args>
55. decltype(auto) apply_visitor(
56.     Visitor&& visitor,
57.     /*const*/ DiscriminatedPtr<Args...>& variant) {
58.     return variant.apply(std::forward<Visitor>(visitor));
59. }

60. template <typename Visitor, typename... Args>
61. decltype(auto) apply_visitor(
62.     Visitor&& visitor,
63.     DiscriminatedPtr<Args...>&& variant) {
64.     return variant.apply(std::forward<Visitor>(visitor));
65. }
```

Safe pointer to one of several types

define, is there the type T in the pack of Types?

```
1.  namespace dptr_detail {

2.  template <typename... Types>
3.  struct GetTypeIndex;

4.  template <typename T, typename... Types>
5.  struct GetTypeIndex<T, T, Types...> {
6.      static const size_t value = 1;
7.  };

8.  template <typename T, typename U, typename... Types>
9.  struct GetTypeIndex<T, U, Types...> {
10.     static const size_t value = 1 + GetTypeIndex<T, Types...>::value;
11. };

12. }
```

Safe pointer to one of several types

invoke the visitor function for a type T* of the DiscriminatedPtr

```
1.  template <size_t index, typename Visitor, typename Result, typename... Types>
2.  struct ApplyVImpl;

3.  template <typename Visitor, typename Result, typename T, typename... Types>
4.  struct ApplyVImpl<1, Visitor, Result, T, Types...> {
5.      Result operator()(size_t, Visitor&& visitor, void* ptr) const {
6.          return visitor(static_cast<T*>(ptr));
7.      }
8.  };
9.  Template <size_t i, typename Visitor, typename Result, typename T, typename... Types>
10. struct ApplyVImpl<i, Visitor, Result, T, Types...> {
11.     Result operator()(size_t runtime_index, Visitor&& visitor, void* ptr) const {
12.         return runtime_index == 1
13.            ? visitor(static_cast<T*>(ptr))
14.            : ApplyVImpl<i-1,Visitor,Result,Types...>{}((runtimeIndex-1),
15.                                                        std::forward<Visitor>(visitor),
16.                                                        ptr);
17.     }
18. };
19. template <typename Visitor, typename... Types>
20. using ApplyVisitor = ApplyVImpl < sizeof...(Types),
21.                                   Visitor,
22.                                   typename VisitorResult<V, Types...>::type,
23.                                   Types... >;
```

Tuple

- Класс ***Tuple*** - контейнер хранения гетерогенных (разнородных) элементов, статически известного размера.
- ***std::tuple*** реализован с помощью ***variadic templates*** и этот класс (не объект!) может содержать любое количество элементов разного типа. Каждый объект хранит конкретный набор элементов.
- Интересен он даже не столько тем, что для его написания и создания вспомогательных функций сейчас используются ***variadic templates***, сколько тем, что ***tuple*** — это *рекурсивная структура данных*.
- Приведём пример простейшей реализации такого класса, которая показывает основную технику работы с ***variadic templates*** — “*откусывание головы*” пака параметров и рекурсивная обработка “хвоста”, которая, кстати, также широко распространена в функциональных языках.

Tuple

рассмотрим функциональность класса кортежа

```
1.  tuple<int, char> t1(42, 'a');
2.  tuple<int, char> t2(t1);
3.  get<0>(t1) = 43;
4.  get<1>(t2) = get<1>(t1) + 1;
5.  get<int>(t2) = 43;                              // get<T> added in C++ 14
6.  set<tuple<int, string>> tuples;                 // OK, tuple has operator<
7.  cout << tuple_size<tuple<int, char>>::value;    // prints '2'

8.  tuple<char const*, int> t3 = make_tuple("Hello", 42);
9.  // heterogenous copy construction
10. tuple<string, int> t4 = t3;
11. tuple<char const*, int, string, int> t5 = tuple_cat(t3, t4);
12. int x;
13. string s;
14. tie(x, s) = make_tuple(42, "hello");            // initialize both x and s
15. tie(ignore, s) = make_tuple(-1, "goodbye");     // initialize only s
```

Tuple

class declaration будет variadic template любого числа типов

```
1.  template <typename... Types>
2.  class tuple;
```

как представить элементы кортежа?

Tuple

class declaration будет variadic template любого числа типов

```
1.  template <typename... Types>
2.  class tuple;
```

как представить элементы кортежа?

наследование?

```
1.  template <typename... Types>
2.  class tuple : Types...
3.  {};
```

Tuple

class declaration будет variadic template любого числа типов

```
1.  template <typename... Types>
2.  class tuple;
```

как представить элементы кортежа?

наследовать кортеж от wrapper type?

```
1.  template <typename T>
2.  struct tuple_element {
3.      T value_;
4.  };

5.  template <typename... Types>
6.  class tuple : tuple_element<Types>...
7.  {};
```

Tuple

class declaration будет variadic template любого числа типов

```
1.  template <typename... Types>
2.  class tuple;
```

как представить элементы кортежа?

наследовать кортеж от wrapper type? Добавим индекс?

```
1.  template <size_t I, typename T>
2.  struct tuple_element {
3.      T value_;
4.  };

5.  template <typename... Types>
6.  class tuple : tuple_element<???, Types>...
7.  {};
```

Tuple

нужно заменить placeholder набором параметров типа size_t

```
1.  template <size_t... Indices, typename... Types>
2.  class tuple : tuple_element<Indices, Types>...
3.  {};
```

может ли тогда отличаться число индексов и число типов?

Tuple

нужно заменить placeholder набором параметров типа size_t

```
1.  template <size_t... Indices, typename... Types>
2.  class tuple : tuple_element<Indices, Types>...
3.  {};
```

может ли тогда отличаться число индексов и число типов?

compilation error

Tuple

нужно заменить placeholder набором параметров типа size_t

```
1.  template <size_t... Indices, typename... Types>
2.  class tuple : tuple_element<Indices, Types>...
3.  {};
```

а использовать как?

```
1.  tuple<0, 1, int, string> tup;
```

Tuple

расцепим класс tuple от индексов, через скрытый класс реализации

```
1.  template <size_t... Indices, typename... Types>
2.  class tuple_impl : tuple_element<Indices, Types>...
3.  {};


4.  template <typename... Types>
5.  class tuple : tuple_impl<???, Types...>
6.  {};
```

как обеспечить tuple_impl всеми необходимыми индексами из tuple?

Tuple

класс, инкапсулирующий набор параметров типа size_t

```
1.  template <size_t... Indices>
2.  struct index_sequence {
3.      using type = index_sequence<Indices...>;
4.  };
```

и что?

Tuple

класс, инкапсулирующий набор параметров типа size_t

```
1.  template <size_t... Indices>
2.  struct index_sequence {
3.      using type = index_sequence<Indices...>;
4.  };
```

как выглядит функция, которая принимает index_sequence и печатает все индексы?

Tuple

класс, инкапсулирующий набор параметров типа size_t

```
1.  template <size_t... Indices>
2.  struct index_sequence {
3.      using type = index_sequence<Indices...>;
4.  };
```

как выглядит функция, которая принимает index_sequence и печатает все индексы?

```
1.  template <size_t... Indices>
2.  void g(index_sequence<Indices...>) {
3.      int ignore[]{ ([](size_t s) { std::cout << s << ' '; }(Indices), 0)... };
4.      (void)ignore; // silence compiler warnings about the unused local variable
5.  }

6.  g(index_sequence<1, 2, 3, 4, 5>{});
```

Tuple

Как индексы появятся в списке по заданному parameter pack (typename… Types)?

```
1.  template <typename Sequence, typename... Types>
2.  struct tuple_impl; // undefined base; parameters are named for clarity only

3.  template <size_t... Indices, typename... Types>
4.  struct tuple_impl<index_sequence<Indices...>, Types...>
5.      : public tuple_element<Indices, Types>...
6.  {
7.  };
```

проитерируемся по index_sequence

Tuple

Как индексы появятся в списке по заданному parameter pack (typename… Types)?

```
1.  template <typename Sequence, typename... Types>
2.  struct tuple_impl; // undefined base; parameters are named for clarity only

3.  template <size_t... Indices, typename... Types>
4.  struct tuple_impl<index_sequence<Indices...>, Types...>
5.      : public tuple_element<Indices, Types>...
6.  {
7.  };
```

проитерируемся по index_sequence

```
1.  template <typename T0, typename T1, typename T2>
2.  class tuple : public tuple_impl<index_sequence<0, 1, 2>, T0, T1, T2>
3.  {};
```

как реализовать make_index_sequence?

Tuple

defined by recursively appending elements to the end of a growing index sequence

```
1.  template <size_t I, typename Sequence>
2.  struct cat_index_sequence;

3.  template <size_t I, size_t... Indices>
4.  struct cat_index_sequence<I, index_sequence<Indices...>>
5.      : index_sequence<Indices..., I>
6.  {};


7.  template <size_t N>
8.  struct make_index_sequence
9.      : cat_index_sequence<N-1, typename make_index_sequence<N-1>::type>::type
10. {};

11. template <>
12. struct make_index_sequence<1> : index_sequence<0> {};
```

Tuple

итоговый класс кортежа будет получен объединением сказанного

```
1.  template <size_t, typename T>
2.  struct tuple_element {
3.      T value_;
4.  };


5.  template <typename Sequences, typename... Types>
6.  struct tuple_impl; // undefined base; parameters are named for clarity only

7.  template <size_t... Indices, typename... Types>
8.  struct tuple_impl<index_sequence<Indices...>, Types...>
9.      : public tuple_element<Indices, Types>...
10. {};


11. template <typename... Types>
12. class tuple : public
13.     tuple_impl<typename make_index_sequence<sizeof...(Types)>::type, Types...>
14. {};
```

Tuple

добавим пару операций в tuple_element и конструкторы tuple

```
1.  explicit tuple_element(T const& value) : value_(value) {}
2.  explicit tuple_element(T&& value) : value_(std::move(value)) {}


3.  template <typename... Types>
4.  class tuple : public
5.      tuple_impl<typename make_index_sequence<sizeof...(Types)>::type, Types...>
6.  {
7.      tuple() = default;
8.      tuple(tuple const&) = default;
9.      tuple(tuple&&) = default;
10.     tuple& operator=(tuple const& rhs) = default;
11.     tuple& operator=(tuple&&) = default;
12. };


13. tuple<int, float> t1(3, 3.14f);          // exact match
14. tuple<long long, double> t2(3, 3.14f);   // widening conversions
15. tuple<string> t3("Hello, World");        // string construction from char const[]
```

Tuple

такой конструктор должен быть шаблоном и принимать forwarding references

```
1.  using base_t = tuple_impl < typename make_index_sequence<sizeof...(Types)>::type,
2.                              Types...
3.                            >;

4.  template <typename... OtherTypes>
5.  explicit tuple(OtherTypes&&... elements)
6.      : base_t(std::forward<OtherTypes>(elements)...)
7.  {}
```

прокидываем параметры конструкторов элементов глубже

```
1.  template <typename... OtherTypes>
2.  explicit tuple_impl(OtherTypes&&... elements)
3.      : tuple_element<Indices, Types>(std::forward<OtherTypes>(elements))...
4.  {}
```

Tuple

обеспечиваем типо-безопасность, чтобы

```
1.  using base_t = tuple_impl < typename make_index_sequence<sizeof...(Types)>::type,
2.                              Types...
3.                            >;

4.  template < typename... OtherTypes,
5.             typename = typename std::enable_if<
6.                 sizeof...(OtherTypes) == sizeof...(Types)>::type >
7.  explicit tuple(OtherTypes&&... elements)
8.      : base_t(std::forward<OtherTypes>(elements)...)
9.  {}

10. // and the same thing applies to tuple_impl's constructor
```

Tuple

обеспечиваем типо-безопасность, чтобы

```
1.  using base_t = tuple_impl < typename make_index_sequence<sizeof...(Types)>::type,
2.                              Types...
3.                            >;

4.  template < typename... OtherTypes,
5.             typename = typename std::enable_if<
6.                 sizeof...(OtherTypes) == sizeof...(Types)>::type >
7.  explicit tuple(OtherTypes&&... elements)
8.      : base_t(std::forward<OtherTypes>(elements)...)
9.  {}

10. // and the same thing applies to tuple_impl's constructor



11. tuple<int> t1(3);   // good, construct from rvalue int
12. tuple<int> t2(t1);  // which constructor are we calling?
13.                     // the template constructor with OtherTypes = tuple<int>&.
```

Tuple

обеспечиваем типо-безопасность, чтобы

```
1.  using base_t = tuple_impl < typename make_index_sequence<sizeof...(Types)>::type,
2.                              Types...
3.                            >;

4.  template < typename... OtherTypes,
5.             typename = typename std::enable_if<
6.                 sizeof...(OtherTypes) == sizeof...(Types)>::type >
7.  explicit tuple(OtherTypes&&... elements)
8.      : base_t(std::forward<OtherTypes>(elements)...)
9.  {}


10. template <typename... OtherTypes>
11. explicit tuple_impl(OtherTypes&&... elements)
12.     : tuple_element<Indices, Types>(std::forward<OtherTypes>(elements))...
13. {}
```

Tuple

tuples of some other types, and naked lists of the tuple's elements

```
1.  template <typename... OtherTypes>
2.  explicit tuple(tuple<OtherTypes...> const& rhs) : base_t(rhs) {}

3.  template <typename... OtherTypes>
4.  explicit tuple(tuple<OtherTypes...>&& rhs) : base_t(std::move(rhs))
    {}
```

реализовать мета-функцию is_tuple_impl, является ли T типом tuple_impl?

Tuple

tuples of some other types, and naked lists of the tuple's elements

```
1.  template <typename... OtherTypes>
2.  explicit tuple(tuple<OtherTypes...> const& rhs) : base_t(rhs) {}

3.  template <typename... OtherTypes>
4.  explicit tuple(tuple<OtherTypes...>&& rhs) : base_t(std::move(rhs))
    {}
```

реализовать мета-функцию is_tuple_impl, является ли T типом tuple_impl?

```
1.  template <typename>
2.  struct is_tuple_impl
3.      : std::false_type {};

4.  template <size_t... Indices, typename... Types>
5.  struct is_tuple_impl<tuple_impl<index_sequence<Indices...>, Types...>>
6.      : std::true_type
7.  {};
```

Tuple

реализовать мета-функцию is_any_of<Op, …Types>

есть ли среди parameter pack …Types хоть 1 тип T, удовлетворяющий операции Op?

Tuple

реализовать мета-функцию is_any_of<Op, …Types>

есть ли среди parameter pack …Types хоть 1 тип T, удовлетворяющий операции Op?

```
1.  template <template <class> typename>
2.  constexpr bool is_any_of() {
3.      return false;
4.  }

5.  template <template <class> typename Op, typename Head, typename... Tail>
6.  constexpr bool is_any_of() {
7.      return Op<Head>::value || is_any_of<Op, Tail...>();
8.  }
```

Tuple

реализовать мета-функцию is_any_of<Op, …Types>

есть ли среди parameter pack …Types хоть 1 тип T, удовлетворяющий операции Op?

```
1.  template <template <class> typename>
2.  constexpr bool is_any_of() {
3.      return false;
4.  }

5.  template <template <class> typename Op, typename Head, typename... Tail>
6.  constexpr bool is_any_of() {
7.      return Op<Head>::value || is_any_of<Op, Tail...>();
8.  }
```

```
9.  template <typename... OtherTypes,
10.     typename = typename std::enable_if<
11.         !is_any_of<is_tuple_impl,
12.         typename std::decay<OtherTypes>::type...>{}>::type >
13. explicit tuple_impl(OtherTypes&&... elements)
14.     : tuple_impl<Indices, Types>(std::forward<OtherTypes>(elements))...
15. {}
```

Tuple

получим элемент из tuple

```
1.  template <size_t I, typename... Types>
2.  ??? get(tuple<Types...> const& tup);

3.  template <size_t I, typename... Types>
4.  ??? get(tuple<Types...>& tup);

5.  template <size_t I, typename... Types>
6.  ??? get(tuple<Types...>&& tup);
```

нужна функция type_at_index, возвращает тип i-го элемента?

Tuple

получим элемент из tuple

```
1.  template <size_t I, typename... Types>
2.  ??? get(tuple<Types...> const& tup);

3.  template <size_t I, typename... Types>
4.  ??? get(tuple<Types...>& tup);

5.  template <size_t I, typename... Types>
6.  ??? get(tuple<Types...>&& tup);
```

нужна функция type_at_index, возвращает тип i-го элемента?

```
1.  template <size_t I, typename Head, typename... Tail>
2.  struct type_at_index {
3.      using type = typename type_at_index<I - 1, Tail...>::type;
4.  };
5.  template <typename Head, typename... Tail>
6.  struct type_at_index<0, Head, Tail...> { using type = Head; };

7.  template <size_t I, typename... Types>
8.  using type_at_index_t = typename type_at_index<I, Types...>::type;
```

Tuple

получим элемент из tuple

```
1.  template <size_t I, typename... Types>
2.  type_at_index<I, Types...> const& get(tuple<Types...> const& tup);

3.  template <size_t I, typename... Types>
4.  type_at_index<I, Types...>& get(tuple<Types...>& tup);

5.  template <size_t I, typename... Types>
6.  type_at_index<I, Types...>&& get(tuple<Types...>&& tup);
```

нужна функция type_at_index, возвращает тип i-го элемента?

```
1.  template <size_t I, typename Head, typename... Tail>
2.  struct type_at_index {
3.      using type = typename type_at_index<I - 1, Tail...>::type;
4.  };
5.  template <typename Head, typename... Tail>
6.  struct type_at_index<0, Head, Tail...> { using type = Head; };

7.  template <size_t I, typename... Types>
8.  using type_at_index_t = typename type_at_index<I, Types...>::type;
```

Tuple

получим элемент из tuple

```
1.  template <size_t I, typename... Types>
2.  type_at_index<I, Types...> const& get(tuple<Types...> const& tup);

3.  template <size_t I, typename... Types>
4.  type_at_index<I, Types...>& get(tuple<Types...>& tup);

5.  template <size_t I, typename... Types>
6.  type_at_index<I, Types...>&& get(tuple<Types...>&& tup);
```

третий прототип неверный, что с ним не так?

Tuple

получим элемент из tuple

```
1.  template <size_t I, typename... Types>
2.  type_at_index<I, Types...> const& get(tuple<Types...> const& tup);

3.  template <size_t I, typename... Types>
4.  type_at_index<I, Types...>& get(tuple<Types...>& tup);

5.  template <size_t I, typename... Types>
6.  type_at_index<I, Types...>&& get(tuple<Types...>&& tup);
```

третий прототип неверный, что с ним не так?

```
1.  int x = 42;
2.  tuple<int&> t(x);
3.  get<0>(std::move(t)); // returns int&!
```

Tuple

получим элемент из tuple

```
1.  template <size_t I, typename... Types>
2.  type_at_index<I, Types...> const& get(tuple<Types...> const& tup);

3.  template <size_t I, typename... Types>
4.  type_at_index<I, Types...>& get(tuple<Types...>& tup);

5.  template <size_t I, typename... Types>
6.  std::remove_reference_t<type_at_index<I, Types...>>&&
7.  get(tuple<Types...>&& tup);
```

третий прототип неверный, что с ним не так?

```
1.  int x = 42;
2.  tuple<int&> t(x);
3.  get<0>(std::move(t)); // returns int&! → int&&
```

Tuple

получим элемент из tuple – реализация функции get

```
1.  template <size_t I, typename... Types>
2.  type_at_index<I, Types...> get(tuple<Types...> const& tup) {
3.      tuple_element<I, type_at_index_t<I, Types...>>& base = tup;
4.      return base.value_;
5.  }
```

Tuple

теперь вторая версия функции get (C++14)

```
1.  tuple<int, string> t1;
2.  get<int>(t1) = 42;
3.  get<float>(t1) = 3.14f;     // compilation error

4.  tuple<int, int> t2;
5.  get<int>(t2) = 42;          // compilation error
```

Tuple

```
1.  template <typename>
2.  constexpr int count() { return 0; }

3.  template <typename T, typename Head, typename... Tail>
4.  constexpr int count() {
5.      return (std::is_same<T, Head>::value ? 1 : 0) + count<T, Tail...>();
6.  }

7.  template <typename>
8.  constexpr int find(int) { return -1; }

9.  template <typename T, typename Head, typename... Tail>
10. constexpr int find(int current_index = 0) {
11.     return std::is_same<T, Head>::value
12.         ? current_index
13.         : find<T, Tail...>(current_index + 1);
14. }

15. template <typename T, typename... Types>
16. T& get(tuple<Types...>& tup) {
17.     static_assert(count<T, Types...>() == 1,
18.                   "T must appear exactly once in ...Types");
19.     return get<find<T, Types...>()>(tup);
20. }
```

Tuple

Implement tuple_size

```
1.  template <typename>
2.  struct tuple_size; // undefined base template

3.  template <typename... Types>
4.  struct tuple_size<tuple<Types...>> : std::integral_constant<size_t,
    sizeof...(Types)>
5.  {};
```

Tuple

forward arguments to get tuple

```
1.  string s, t;
2.  auto t1 = forward_as_tuple(s, t);              // t1 is tuple<string&, string&>
3.  auto t2 = forward_as_tuple(std::move(s), t);   // t2 is tuple<string&&, string&>
```

Tuple

tie функция-helper очень полезна, чтобы связывать в tuple отдельные переменные

```
1.  int status_code; string status;
2.  tie(status_code, status) = make_http_request(url);

3.  set<string> names;
4.  bool inserted; set<string>::iterator iter;
5.  tie(inserted, iter) = names.insert("Sasha");
6.  tie(inserted, ignore) = names.insert("Sasha"); // assigns only to 'inserted'
```

Tuple

tie функция-helper очень полезна, чтобы связывать в tuple отдельные переменные

```
1.  int status_code; string status;
2.  tie(status_code, status) = make_http_request(url);

3.  set<string> names;
4.  bool inserted; set<string>::iterator iter;
5.  tie(inserted, iter) = names.insert("Sasha");
6.  tie(inserted, ignore) = names.insert("Sasha"); // assigns only to 'inserted'
```

```
7.  template <typename... Types>
8.  tuple<Types&...> tie(Types&... elements) {
9.      return tuple<Types&...>(elements...);
10. }
```

Tuple

tie функция-helper очень полезна, чтобы связывать в tuple отдельные переменные

```
1.  int status_code; string status;
2.  tie(status_code, status) = make_http_request(url);

3.  set<string> names;
4.  bool inserted; set<string>::iterator iter;
5.  tie(inserted, iter) = names.insert("Sasha");
6.  tie(inserted, ignore) = names.insert("Sasha"); // assigns only to 'inserted'
```

```
7.  template <typename... Types>
8.  tuple<Types&...> tie(Types&... elements) {
9.      return tuple<Types&...>(elements...);
10. }

11. struct ignore_t {
12.     template <typename U>
13.     ignore_t& operator=(U&&) { return *this; }
14. } ignore;
```

Tuple

tuple_cat – конкатенация кортежей

```
1.  auto t1 = tuple_cat(make_tuple(1), make_tuple(2)); // tuple<int, int>(1, 2)
2.  auto t2 = tuple_cat(t1, make_tuple(3), make_tuple(4));
3.  auto t3 = tuple_cat(t1, t1, t2, t2);
4.  // t3 is a tuple of 12 ints: 1, 2, 1, 2, 1, 2, 3, 4, 1, 2, 3, 4
```

Tuple

tuple_cat – конкатенация кортежей

```
1.  auto t1 = tuple_cat(make_tuple(1), make_tuple(2)); // tuple<int, int>(1, 2)
2.  auto t2 = tuple_cat(t1, make_tuple(3), make_tuple(4));
3.  auto t3 = tuple_cat(t1, t1, t2, t2);
4.  // t3 is a tuple of 12 ints: 1, 2, 1, 2, 1, 2, 3, 4, 1, 2, 3, 4
```

explode разрушает tuple на составляющие и передаёт его элементы в function?

Tuple

tuple_cat – конкатенация кортежей

```
1.  auto t1 = tuple_cat(make_tuple(1), make_tuple(2)); // tuple<int, int>(1, 2)
2.  auto t2 = tuple_cat(t1, make_tuple(3), make_tuple(4));
3.  auto t3 = tuple_cat(t1, t1, t2, t2);
4.  // t3 is a tuple of 12 ints: 1, 2, 1, 2, 1, 2, 3, 4, 1, 2, 3, 4
```

explode разрушает tuple на составляющие и передаёт его элементы в function?

```
5.  template <typename Tuple, size_t... Indices>
6.  void explode1(Tuple&& tup, index_sequence<Indices...>) {
7.      function(get<Indices>(std::forward<Tuple>(tup))...);
8.  }

9.  template <typename Tuple>
10. void explode(Tuple&& tup) {
11.     explode1(std::forward<Tuple>(tup),
12.              make_index_sequence<tuple_size<std::decay_t<Tuple>>::value>{});
13. }
```

Tuple

tuple_cat – конкатенация кортежей

```
1.  auto t1 = tuple_cat(make_tuple(1), make_tuple(2)); // tuple<int, int>(1, 2)
2.  auto t2 = tuple_cat(t1, make_tuple(3), make_tuple(4));
3.  auto t3 = tuple_cat(t1, t1, t2, t2);
4.  // t3 is a tuple of 12 ints: 1, 2, 1, 2, 1, 2, 3, 4, 1, 2, 3, 4
```

написать tuple_cat?

```
5.  template <typename Tuple, size_t... Indices>
6.  void explode1(Tuple&& tup, index_sequence<Indices...>) {
7.      function(get<Indices>(std::forward<Tuple>(tup))...);
8.  }

9.  template <typename Tuple>
10. void explode(Tuple&& tup) {
11.     explode1(std::forward<Tuple>(tup),
12.              make_index_sequence<tuple_size<std::decay_t<Tuple>>::value>{});
13. }
```

Tuple

tuple_cat – конкатенация кортежей

```
1.  template <typename Tuple1, size_t... Indices1, typename Tuple2, size_t... Indices2>
2.  auto tuple_cat1(Tuple1&& tup1, Tuple2&& tup2,
3.                  index_sequence<Indices1...>, index_sequence<Indices2...>) {
4.      return make_tuple(
5.          get<Indices1>(std::forward<Tuple1>(tup1))...,
6.          get<Indices2>(std::forward<Tuple2>(tup2))...
7.      );
8.  }

9.  template <typename Tuple1, typename Tuple2>
10. auto tuple_cat(Tuple1&& tup1, Tuple2&& tup2) {
11.     return tuple_cat1(
12.         std::forward<Tuple1>(tup1),
13.         std::forward<Tuple2>(tup2),
14.         make_index_sequence<tuple_size<std::decay_t<Tuple1>>::value>{},
15.         make_index_sequence<tuple_size<std::decay_t<Tuple2>>::value>{}
16.     );
17. }
```

```
1.  template <size_t, typename>
2.  struct type_at_tuple; // undefined base template

3.  template <size_t I, typename... Types>
4.  struct type_at_tuple<I, tuple<Types...>> : type_at_index<I, Types...>
5.  {};


6.  template <typename, typename, typename, typename>
7.  struct cat1_t; // undefined base template

8.  template <typename Tuple1, size_t... Indices1, typename Tuple2, size_t... Indices2>
9.  struct cat1_t < Tuple1, index_sequence<Indices1...>,
10.                 Tuple2, index_sequence<Indices2...>
11.               > {
12.     using type = tuple < typename type_at_tuple<Indices1, Tuple1>::type...,
13.                          typename type_at_tuple<Indices2, Tuple2>::type...
14.                        >;
15. };

16. template <typename Tuple1, typename Tuple2>
17. struct cat_t {
18.     static constexpr size_t Size1 = tuple_size<std::decay_t<Tuple1>>::value;
19.     static constexpr size_t Size2 = tuple_size<std::decay_t<Tuple2>>::value;
20.     using Seq1 = typename make_index_sequence<Size1>::type;
21.     using Seq2 = typename make_index_sequence<Size2>::type;

22.     using type = typename cat1_t<Tuple1, Seq1, Tuple2, Seq2>::type;
23. };
```

Tuple

обобщим tuple_cat на любое количество tuple-ов

```
1.  template <typename HeadTuple>
2.  HeadTuple&& tuple_cat(HeadTuple&& tup) {
3.      return std::forward<HeadTuple>(tup);
4.  }


5.  template <typename Head1Tuple, typename Head2Tuple, typename... TailTuples>
6.  auto tuple_cat(Head1Tuple&& tup1, Head2Tuple&& tup2, TailTuples&&... tail) {
7.      return tuple_cat(
8.          tuple_cat2(
9.              std::forward<Head1Tuple>(tup1),
10.             std::forward<Head2Tuple>(tup2)
11.         ),
12.         std::forward<TailTuples>(tail)...
13.     );
14. }
```

```
1.  template <typename, typename>
2.  struct cat_two_t; // undefined base template

3.  template <typename... Types1, typename... Types2>
4.  struct cat_two_t<tuple<Types1...>, tuple<Types2...>> {
5.      using type = tuple<Types1..., Types2...>;
6.  };


7.  template <typename...>
8.  struct cat_many_t; // undefined base template

9.  template <typename Tuple>
10. struct cat_many_t<Tuple> {
11.     using type = Tuple;
12. };

13. template <typename Tuple1, typename Tuple2, typename... Tuples>
14. struct cat_many_t<Tuple1, Tuple2, Tuples...> {
15.     using two_t = typename cat_two_t<Tuple1, Tuple2>::type;
16.     using type = typename cat_many_t<two_t, Tuples...>::type;
17. };
```

Tuple

tuple_cat быстрый (Sasha Goldshtein + Stephan T. Lavavej)

```
1.    just one line of code. But it’s an ingenious line of code.
2.   return make_tuple(get<Jx>(get<Ix>(forward<Tuples>(tuples)))...);
```

Tuple

tuple_cat быстрый (Sasha Goldshtein + Stephan T. Lavavej)

```
1.   just one line of code. But it's an ingenious line of code.

2.   return make_tuple(get<Jx>(get<Ix>(forward<Tuples>(tuples)))...);
```

```
3.   tuple<int, char>,
4.   tuple<float>,
5.   tuple<char, bool, long>

6.   tuple<tuple<int, char>, tuple<float>, tuple<char, bool, long>>
```

```
7.   Ix = [0, 0, 1, 2, 2, 2]
8.   Jx = [0, 1, 0, 0, 1, 2]
```

Tuple

tuple_cat быстрый (Sasha Goldshtein + Stephan T. Lavavej)

```
1.   just one line of code. But it's an ingenious line of code.

2.   return make_tuple(get<Jx>(get<Ix>(forward<Tuples>(tuples)))...);



3.   tuple<int, char>,
4.   tuple<float>,
5.   tuple<char, bool, long>

6.   tuple<tuple<int, char>, tuple<float>, tuple<char, bool, long>>


7.   Ix = [0, 0, 1, 2, 2, 2]
8.   Jx = [0, 1, 0, 0, 1, 2]

9.   make_tuple(
10.      get<0>(get<0>(tuples)), get<1>(get<0>(tuples)),
11.      get<0>(get<1>(tuples)),
12.      get<0>(get<2>(tuples)), get<1>(get<2>(tuples)), get<2>(get<2>(tuples))
13.  );
```

Tuple

tuple_cat быстрый (Sasha Goldshtein + Stephan T. Lavavej)

```
1.   just one line of code. But it's an ingenious line of code.

2.   return make_tuple(get<Jx>(get<Ix>(forward<Tuples>(tuples)))...);



3.   tuple<int, char>,
4.   tuple<float>,
5.   tuple<char, bool, long>

6.   tuple<tuple<int, char>, tuple<float>, tuple<char, bool, long>>


7.   Ix = [0, 0, 1, 2, 2, 2]
8.   Jx = [0, 1, 0, 0, 1, 2]

9.   S(0) times                S(1) times                  S(n-1) times
10.  Ix = [0, 0, ..., 0,       1, 1, ..., 1,       ...,  n-1, n-1, ..., n-1     ]
11.  Jx = [0, 1, ..., S(0)-1,  0, 1, ..., S(1)-1,  ...,  0 ,  1  , ..., S(n-1)-1]
```

Tuple

tuple_cat быстрый (Sasha Goldshtein + Stephan T. Lavavej)

```
1.  template <typename... Tuples, size_t... Ix, size_t... Jx>
2.  auto cat2d(Tuples&& tuples, index_sequence<Ix...>, index_sequence<Jx...>) {
3.      return make_tuple(get<Jx>(get<Ix>(std::forward<Tuples>(tuples)))...);
4.  }


5.  template <typename... Tuples>
6.  auto tuple_cat(Tuples&&... tuples) {
7.      auto ixs = repeating_index_sequence<
8.          std::integral_constant<size_t, 0>,
9.          tuple_size<std::decay_t<Tuples>::type>::value... >{};
10.     auto jxs = variable_index_sequence<
11.         tuple_size<std::decay_t<Tuples>::type>::value... >{};
12.     return cat2d(forward_as_tuple(std::forward<Tuples>(tuples)...), ixs, jxs);
13. }
```

Tuple

tuple_cat быстрый (Sasha Goldshtein + Stephan T. Lavavej)

```
1.  template <typename... Tuples, size_t... Ix, size_t... Jx>
2.  auto cat2d(Tuples&& tuples, index_sequence<Ix...>, index_sequence<Jx...>) {
3.      return make_tuple(get<Jx>(get<Ix>(std::forward<Tuples>(tuples)))...);
4.  }


5.  template <typename... Tuples>
6.  auto tuple_cat(Tuples&&... tuples) {
7.      auto ixs = repeating_index_sequence<
8.          std::integral_constant<size_t, 0>,
9.          tuple_size<std::decay_t<Tuples>::type>::value... >{};
10.     auto jxs = variable_index_sequence<
11.         tuple_size<std::decay_t<Tuples>::type>::value... >{};
12.     return cat2d(forward_as_tuple(std::forward<Tuples>(tuples)...), ixs, jxs);
13. }
```

repeating_index_sequence metafunction & variable_index_sequence metafunction

Tuple

1. repeating_index_sequence metafunction, берёт на вход интегральную константу 0, как начальное значени и размеры всех tuple-ов и генерирует последовательность Ix?

2. repeating_index_sequence<std::integral_constant<size_t, 0>, 2, 1, 3>::type
3. is
4. index_sequence<0, 0, 1, 2, 2, 2>.

5. variable_index_sequence metafunction,
6. получает размеры всех tuple-ов и создаёт последовательности Jx?

7. variable_index_sequence<2, 1, 3>::type
8. is
9. index_sequence<0, 1, 0, 0, 1, 2>

Tuple

```
1.  template <typename...>
2.  struct cat_index_sequence;


3.  template <size_t... Indices>
4.  struct cat_index_sequence<index_sequence<Indices...>> : index_sequence<Indices...>
5.  {};


6.  template <size_t... Indices1, size_t... Indices2>
7.  struct cat_index_sequence<index_sequence<Indices1...>, index_sequence<Indices2...>>
8.      : index_sequence<Indices1..., Indices2...>
9.  {};


10. template <size_t... Indices1, size_t... Indices2, typename... Sequences>
11. struct cat_index_sequence < index_sequence<Indices1...>,
12.                             index_sequence<Indices2...>,
13.                             Sequences...
14.                           >
15. : cat_index_sequence < index_sequence<Indices1..., Indices2...>,
16.                        cat_index_sequence<Sequences...>
17.                      >
18. {};
```

Tuple

```
1.  template <size_t I, size_t Length>
2.  struct make_repeated_sequence : cat_index_sequence<
3.      index_sequence<I>,
4.      typename make_repeated_sequence<I, Length - 1>::type >
5.  {};


6.  template <size_t I>
7.  struct make_repeated_sequence<I, 1> : index_sequence<I>
8.  {};
```

Tuple

```
1.  template <typename, size_t...>
2.  struct repeating_index_sequence;


3.  template <size_t I, size_t Length>
4.  struct repeating_index_sequence<std::integral_constant<size_t, I>, Length>
5.      : make_repeated_sequence<I, Length>
6.  {};


7.  template <size_t I, size_t Length, size_t... TailLengths>
8.  struct repeating_index_sequence < std::integral_constant<size_t, I>,
9.                                    Length,
10.                                   TailLengths...
11.                                >
12.     : cat_index_sequence<
13.           typename make_repeated_sequence<I, Length>::type,
14.           typename repeating_index_sequence < std::integral_constant<size_t, I + 1>,
15.                                               TailLengths...
16.                                             >::type >
17. {};
```

Tuple

1. repeating_index_sequence metafunction, берёт на вход интегральную константу 0, как начальное значени и размеры всех tuple-ов и генерирует последовательность Ix?

2. repeating_index_sequence<std::integral_constant<size_t, 0>, 2, 1, 3>::type
3. is
4. index_sequence<0, 0, 1, 2, 2, 2>.

5. variable_index_sequence metafunction,
6. получает размеры всех tuple-ов и создаёт последовательности Jx?

7. variable_index_sequence<2, 1, 3>::type
8. is
9. index_sequence<0, 1, 0, 0, 1, 2>

Tuple

```
1.  template <size_t...>
2.  struct variable_index_sequence;


3.  template <size_t Head>
4.  struct variable_index_sequence : make_index_sequence<Head>
5.  {};


6.  template <size_t Head, size_t... Tail>
7.  struct variable_index_sequence<Head, Tail...>
8.      : cat_index_sequence < typename variable_index_sequence<Head>::type,
9.                             typename variable_index_sequence<Tail...>::type
10.                          >
11. {};
```

Tuple

Times are relative to simple tuple/simple cat, which is an arbitrary baseline of 100

	simple cat	**two-dimensional cat**
simple tuples	100	71
complex tuples	193	88

std::pair: размещение (emplace)

```
1.  template<typename T1, typename T2> class pair {
2.      // Конструктор, распаковывающий кортеж
3.      template<typename ... ArgTypes1, size_t ... Indices1,
4.               typename ... ArgTypes2, size_t ... Indices2>
5.      pair(std::tuple<ArgTypes1...>& first_args, std::tuple<ArgTypes2...>& second_args,
6.           PackIndices<Indices1...>, PackIndices<Indices2...>) :
7.          first(std::forward<ArgTypes1>(std::get<Indices1>(first_args))...),
8.          second(std::forward<ArgTypes2>(std::get<Indices2>(second_args))...)
9.      {}
10. public:
11.     // Конструктор, доступный пользователю
12.     template<typename ... ArgTypes1, typename ... ArgTypes2>
13.     pair(std::piecewise_construct_t, std::tuple<ArgTypes1...> first_args,
14.                                      std::tuple<ArgTypes2...> second_args) :
15.         pair(first_args, second_args,
16.              typename CreatePackIndices<sizeof ... (ArgTypes1)>::type(),
17.              typename CreatePackIndices<sizeof ... (ArgTypes2)>::type())
18.     {}
19. private:
20.     T1 first;
21.     T2 second;
22. };
```

```cpp
// C++98:
template <typename T1, typename T2>
std::pair<T1, T2>
make_pair(T1 const & t1, T2 const & t2) {
    return std::pair<T1, T2>(t1, t2);
}

// C++03:
template <typename T1, typename T2>
std::pair<T1, T2>
make_pair(T1 t1, T2 t2) {
    return std::pair<T1, T2>(t1, t2);
}

// C++11:
template <typename T1, typename T2>
constexpr
std::pair<typename std::decay<T1>::type, typename std::decay<T2>::type>
make_pair(T1 && t1, T2 && t2) {
    return std::pair<typename std::decay<T1>::type,
        typename std::decay<T2>::type>(std::forward<T1>(t1),
std::forward<T2>(t2));
}

// C++14:
template <typename T1, typename T2>
constexpr
std::pair<std::decay_t<T1>, std::decay_t<T2>>
make_pair(T1 && t1, T2 && t2) {
    return std::pair<std::decay_t<T1>,
        std::decay_t<T2>>(std::forward<T1>(t1), std::forward<T2>(t2));
}
```

Упражнение 1

Type save printf

```
template<typename... Args>
void printf(Args&&... args) { ... }
```

используя стандартную printf, без streams

первым аргументом не идёт строка с процентами

Упражнение 2

- написать compile-time функцию, которая складывает все аргументы, которые в неё подают, в массив и возвращает его

Упражнение 3

- Реализация с помощью variadic templates
- **std::function<>**
- **std::mem_fn<>**

Упражнение 4

Для нашего выведенного класса **tuple** написать

- Copy and move assignment operators
- Comparison operators: ==, !=, <, <=, >, >=
- Constructor of **tuple** from **std::pair** and **std::array**
- Function **make_tuple**()
  (возможно понадобится **std::reference_wrapper**)

СПАСИБО ЗА ВНИМАНИЕ!