

СЕМИНАР 2.

Вопросы по предыдущему семинару

???

Техники программирования на шаблонах

**Немножко разминки (не в зачёт
дорешек, live coding)**

рекурсия

Напишите факториал с помощью параметров шаблонов

```
template<int N> struct F {  
    static constexpr int value = ???;  
}  
  
static constexpr int F5 = F<5>::value;
```

using

Напишите фасад к умным указателям.

```
template<class T>
class PointerToObject {};

template<class T, size_t N>
class PointerToArray {}; // T[N]

template<class T>
class PointerToUnboundArray {}; // T[]

template<class T> using Pointer = ???;
// выберет одну из реализаций
```

ВЕТВЛЕНИЕ

Напишите функцию, находящую тип с максимальным размером

```
template<class A, class B> class Maximal {  
    using type = ???  
};
```

(см. `std::conditional_t`)

Задачи

1. Метапрограммирование на типах

[1.5 + 0.5 балла]

Множество - это набор уникальных значений (элементов).
Порядок перечисления элементов значения не имеет.
Множества идентичны, если они содержат одни и те же элементы.

Давайте напишем библиотечку для множества типов.

Которая пригодится нам, например, для того, чтобы порождать ``std::variant`` ы с нужными наборами типов.

Вопрос: сможем ли мы порождать `std::tuple` ?

Основная часть [1.5 балла]

API, которое нужно сделать

```
// наше множество
template<class...> struct typeset {};

// проверка на пустоту
template<class Typeset> constexpr bool is_empty = ???;
// мощность
template<class Typeset> constexpr size_t size = ???;
// извлекаем элемент множества
template<size_t N, class Typeset> using get = ???;

// проверяем вхождение в множество
template<class T, class Typeset>
constexpr bool belongs_to = ???;
```

```
// объединение, пересечение, асимметричная разность
template<class Ts1, class Ts2>
using join = ???;
template<class Ts1, class Ts2>
using cross = typeset<>;
template<class Ts1, class Ts2>
using subtract = typeset<>;

// порождаем std::variant
template<class Typeset>
using variant_of = std::variant<???>;
```

что нам здесь пригодится:

Хедер `<type_traits>`

- **метафункции** `std::is_same`, `std::conditional_t`
- **метаконстанты** `std::integer_constant`, `std::bool_constant`,
`std::true_type`, `std::false_type`

Техника alias + реализация на специализациях

```
template<class> struct impl {};  
// определяет внутри какой-нибудь type или value  
  
template<.....> struct impl<.....> {};  
template<.....> struct impl<.....> {};  
  
template<class T> using facade = impl<T>::type;  
// если нет подходящей специализации,  
// получим substitution failure
```

Итерирование по variadic parameters - сопоставление с образцом

```
template<class FixedArg, class...> struct impl;  
  
template<class FixedArg>  
struct impl<FixedArg> {};  
// для случая пустого списка  
  
template<class FixedArg, class T1, class... Ts>  
struct impl<FixedArg, T1, Ts...> {};  
// сопоставляем список  
// с первым параметром (T1) и хвостом (Ts...)
```

Сложные формы сопоставления

```
template<class T> struct impl;  
template<class... Args> struct impl<typeset<Args...>>;
```

Ну и оператор `sizeof`...

Проверка уникальности. [0.5 балла]

Сделайте проверку, чтобы нельзя было инстанцировать `typeset` с повторяющимися элементами.

2. SFINAE [1 балл]

Позволяет уточнять выбор перегрузки или специализации, когда обычного сопоставления недостаточно.

Напишем функцию `pretty_printer`, которая выводит значение в `std::ostream`

```
template<class T>  
void pretty_printer(std::ostream& ost, const T& value);
```

pretty_printer

- для целых чисел - пишет аннотацию типа "signed/unsigned int_8/16/.... VALUE"
- для вещественных - "float/double VALUE_IN_FIXED_FORMAT" (используйте `std::fixed`)
- для типов с определённым `operator<<` - "some printable VALUE"
- для тривиальных типов (POD) - "N bytes B1 B2 ... BN" (в десятичном формате)
- для произвольных типов - "some N bytes"
- для неполных - "incomplete"

(не нужно различать между собой типы `char` / `signed char` / `int8_t`: просто знаковость и размер в битах)

Что нам понадобится:

`<type_traits>` с **метафункциями** `std::is_integral`,
`std::is_floating_point`, `std::is_standard_layout`

А также метафункции `std::enable_if_t` и, **ВОЗМОЖНО**,
`std::void_t`

Написать метафункции `is_printable` и `is_complete`

Сделать адаптер к `is_standard_layout`, чтобы он не давал ошибку компиляции на неполных типах.

Приём SFINAE - попытка узнать тип выражения, зависящий от параметра `decltype(~~~ T ~~~)`

```
template<class T, class DUMMY = void>
struct Foo {.....} // основной шаблон

template<class T>
struct Foo<T, std::void_t<????>> {.....};
// эта специализация подойдёт,
// только если ???? существует и это тип

template<class T>
std::enable_if_t<CONDITION, ReturnType> foo(.....);
// специализация подойдёт (и подставит ReturnType),
// только если CONDITION - true
```

ВНИМАНИЕ! В этой задаче не пользуйтесь `if constexpr`.

Вспомогательная функция для печати в потоке вывода [0.5 балла]

Напишите функцию (или класс, или что угодно на ваш вкус)

`pretty`

```
std::cout << "this is " << pretty(123) << " value!"  
          << std::endl;
```

3. Expression Template [0.5 + 1 балл]

Перегрузка "обычных" операторов и/или функций, творящая магию с типами результатов.

Мы строим некоторое синтаксическое дерево (выражение) и производим вычисления прямо во время компиляции - выполняем свёртку выражения.

Результатом свёртки может быть этажерка типов, которые являются параметрами друг друга. А может быть и упрощение этой этажерки.

Основная часть [0.5 балла]

Напишем библиотеку проекции индексов массива - что-то, похожее на `std::string_view`.

```
// семейство проекций диапазонов [0..n]
// на произвольные индексы
template<size_t From, size_t To>
struct range;

template<class M1, class M2>
struct concat; // M1{} + M2{}

// у которых есть члены
struct some_mapping {
    // граница области определения
    static constexpr size_t size();
    // функция пересчёта индекса
    constexpr size_t operator()(size_t index) const;
};
```



```
range<10, 30>::size() == 30-10 == 20  
range<10, 30>()(5) == 10+5 == 15
```

```
auto a = range<10, 30>() + range<40, 70>(); // concat  
a.size() == 20 + 30 == 50  
a(5) == 5+10 = 15  
a(25) == 25-20+40 = 45
```

Свойства:

- `range` - это линейная функция, определённая на отрезке от 0 до длины диапазона
- `concat` - это кусочно-линейная функция.
- вне области определения все функции возвращают "сигнальное" значение `npos = size_t(-1)`

Обратите внимание: - у объектов нет членов-данных, все их свойства - исключительно в параметрах шаблона. - оператор сложения - `constexpr`.

Оптимизация выражения [+1 балл]

- конкатенация смежных диапазонов - один диапазон
- конкатенация конкатенаций со смежными диапазонами - склеивает смежные части
- конкатенации автоматически пере-расставляют скобки:
 $(a+b)+(c+d) \Rightarrow (((a+b)+c)+d)$

```
range<10,20>( ) + range<20,30>( )  
  
==  
  
range<10,                30>( )
```

```
(range<10, 20>() + range<30, 40>())  
+  
(range<40, 50>() + range<60, 70>())  
==  
concat<  
    concat<  
        range<10, 20>,  
        range<30, 50> // склеенный диапазон  
    >,  
    range<60, 70>  
>()
```

Нам понадобятся перегрузки оператора

- базовый - строит конкатенацию из произвольных проекций
- для смежных диапазонов
- для случая, когда справа конкатенация - $a + (b+c)$
- для случая, когда слева конкатенация, а справа смежный к ней диапазон $(a+r1) + r2$