

Шаблоны с переменным числом аргументов

(Variadic templates)



Проблема

Хотим написать функцию наподобие `std::make_unique` или `std::printf`, принимающие произвольное число аргументов.

```
std::printf("%d %f %s", 1, 2.0, "aba");
```

```
struct A {  
    A(int x = 0, double y = 0.0, std::string z = "") { ... }  
    // ...  
};
```

```
auto a = std::make_unique<A>();  
auto b = std::make_unique<A>(1);  
auto c = std::make_unique<A>(1, 2.0);  
auto d = std::make_unique<A>(1, 2.0, "aba");
```

Variadic functions (C-style)

Для обозначения функции, принимающей произвольное число аргументов, в языке C использовался специальный аргумент - `...` (ellipsis)

```
#include <cstdarg>

void PrintInts(std::ostream& os, int n, ...) {
    std::va_list args; va_start(args, n);
    for (int i = 0; i < n; ++i) {
        os << va_arg(args, int) << (i == n - 1 ? "" : " ");
    }
    va_end(args);
}

float SumDoubles(int n, ...) {
    std::va_list args; va_start(args, n);
    auto res = 0.0;
    for (int i = 0; i < n; ++i) { res += va_arg(args, double); }
    va_end(args);
    return res;
}
```

Variadic templates (C++11)

C++ предоставляет более безопасный интерфейс для работы с функциями с переменным числом аргументов.

Синтаксис:

```
template <class... Args> // или template <typename... Args>
void Print(std::ostream& os, Args... args);

template <class... Args>
auto Sum(Args... args);
```

Args - пакет параметров-типов (type template parameter pack)

args - пакет параметров функции (function parameter pack)

```
Print(std::cout, 0, "a"); // Args = [int, const char*], args = [0, "a"]
Sum(0, 1.0, 1u);         // Args = [int, double, unsigned], args = [0, 1.0, 1u]
```

Variadic templates (C++11)

- Пакеты параметров можно использовать и в шаблонах классов, но только в качестве последнего параметра.

```
template <class T, class... Args> class MyClass { /* ... */ };  
// template <class... Args, class T> class MyClass; - так нельзя
```

```
MyClass<int> x; // T = int, Args = []  
MyClass<int, char, float> y; // T = int, Args = [char, float]
```

- В случае шаблонов функций пакеты параметров могут идти как в начале, так и в конце. Но лучше всегда писать в конце, иначе возможны проблемы.

```
template <class... Args, class T>  
void Function(T x, Args... args); // ok  
  
template <class... Args, class T>  
void Function(Args... args, T x); // ok, но воспользоваться не получится...
```

Variadic templates (C++11)

```
template <class T, class... Args> void Function(T x, Args... args);
```

```
Function(0, 0.0);           // ok: T=int, Args=[double,]  
Function<int, int, int>(0, 0.0, 1); // ok: T=int, Args=[int, int]
```

```
template <class... Args, class T> void Function(T x, Args... args);
```

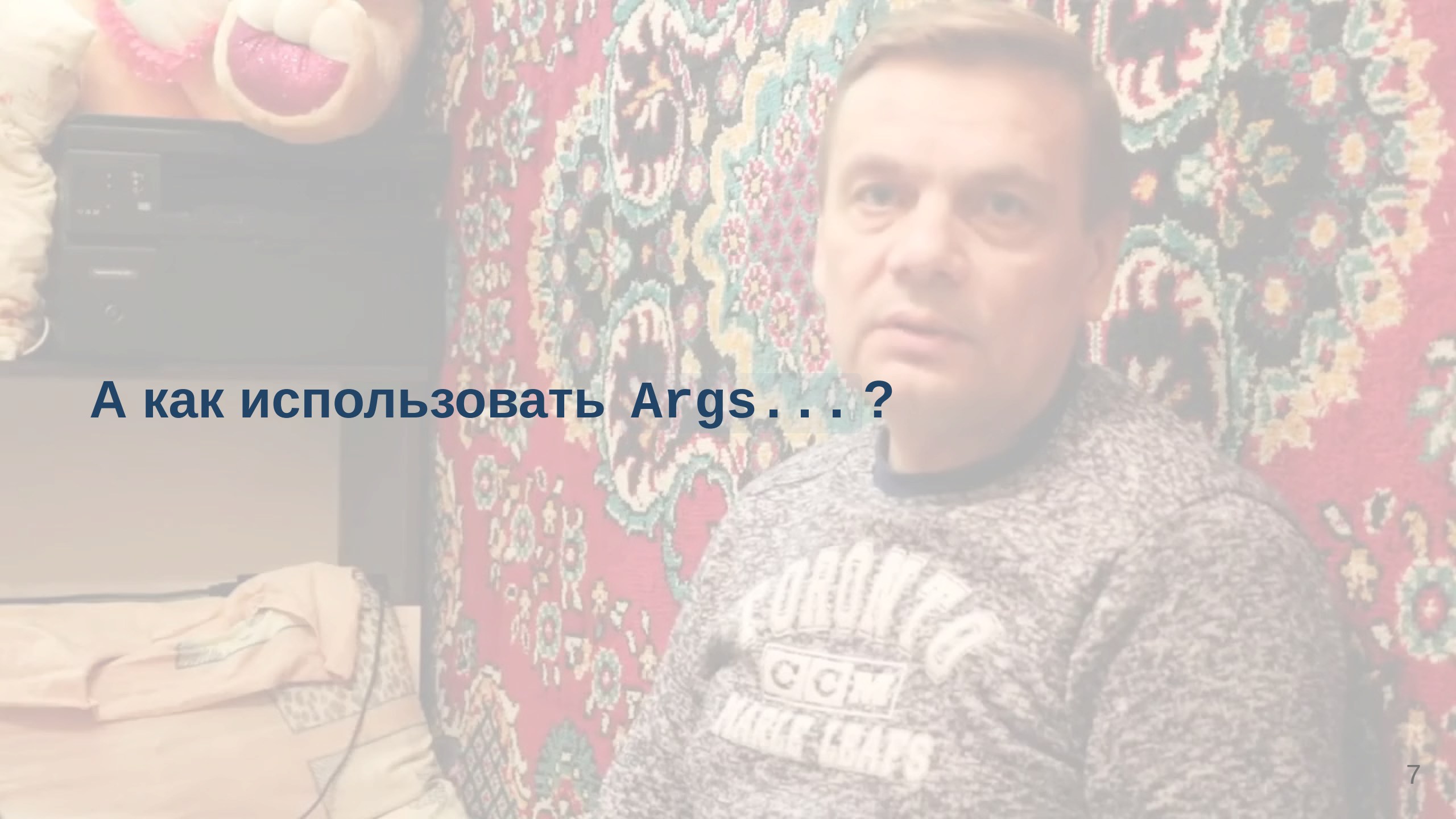
```
Function(0, 0.0);           // ok: Args=[double,], T=int  
// Function<int, int, int>(0, 0.0, 1); // Fail: Args=[int, int, int], T=?
```

```
template <class T, class... Args> void Function(Args... args, T x);
```

```
// Function(0, 0.0);           // Fail: Args=[int, double], T=?  
Function<int, int, int>(0, 0.0, 1); // ok: T=int, Args=[int, int]
```

```
template <class... Args, class T> void Function(Args... args, T x);
```

```
// Function(0, 0.0);           // Fail  
// Function<int, int, int>(0, 0.0, 1); // Fail
```

A man with short brown hair, wearing a grey sweatshirt with "TORONTO CCM HOCKEY LEAPS" printed on it, is looking slightly to the right. He is sitting in front of a red background with a dense floral pattern. In the top left corner, there is a plush toy with a large open mouth showing pink tongue and white teeth. A black electronic device is partially visible behind the plush toy. In the bottom left corner, there is a white object, possibly a bag or a piece of clothing, with some text on it.

А как использовать Args . . . ?

Оператор `sizeof...`

`sizeof...` применяется к пакету параметров и возвращает *количество* элементов в пакете (именно **количество**, а не размер в байтах).

```
template <class... Args>
void Function(const Args&... args) {
    // ...
    std::cout << sizeof...(Args) << '\n';
    std::cout << sizeof...(args) << '\n';
    // ...
}

Function(1, 2, 3);
```

3 3

Использование 1. Распаковка пакета

Если у вас уже есть функция с переменным числом аргументов, то в нее можно передать пакет параметров. Для этого в месте использования необходимо к его имени добавить `...`.

```
template <class... Args>
void PrintTitle(std::ostream& os, std::string_view title, Args... args) {
    os << title << '\n';
    Print(os, args...); // <=> Print(os, args[0], ..., args[n-1]);
}
```

Также при распаковке к пакету можно применить некоторый паттерн, который будет применен к каждому элементу пакета (`pattern(args)...`):

```
template <class... Args>
auto SumSquares(Args... args) {
    return Sum(args * args...);
    // <=> Sum(args[0] * args[0], ..., args[n-1] * args[n-1])
}
```

Использование 1. Распаковка пакета

Еще примеры

```
// принимаем все аргументы по константной ссылке
template <class... Args>
void PrintAddresses(std::ostream& os, const Args&... args) {
    Print(os, &args...);
    // <=> Print(os, &args[0], &args[1], ..., &args[n-1])
}
```

```
template <class... Args>
void PrintPartSums(std::ostream& os, const Args&... args) {
    Print(os, Sum(args...) - args...);
    // <=> Print(os, S - args[0], S - args[1], ..., S - args[n-1])
}
```

Использование 2. Метод "откусывания"

Нельзя просто так взять и обратиться к элементу пакета параметров.

Общий способ работы - обрабатывать элементы по одному и использовать рекурсию.

```
template <class Head> // конец рекурсии (Args = [])
void Print(std::ostream& os, Head head) {
    os << head;
}

template <class Head, class... Args> // "откусили" Head
void Print(std::ostream& os, Head head, Args... args) {
    os << head << ' ';
    Print(os, args...); // передаем аргументы из 'args' дальше
}
```

```
Print(std::cout, 1, "is greater than", 0.0); // "1 is greater than 0"
```

Использование 2. Метод "откусывания"

Нельзя просто так взять и обратиться к элементу пакета параметров.

Общий способ работы - обрабатывать элементы по одному и использовать рекурсию.

```
template <class Head> // конец рекурсии (Args = [])
auto Sum(Head head) {
    return head;
}

template <class Head, class... Args> // "откусили" Head
auto Sum(Head head, Args... args) {
    return head + Sum(args...); // передаем аргументы из 'args' дальше
}
```

```
Sum(1, 1.5, 0l); // 2.5
```

Использование 3. Fold expression (C++17)

Fold expression (выражение свертки) - более удобный интерфейс использования пакетов параметров, позволяющий в большинстве ситуаций обойтись без рекурсии

Синтаксис:

op - некоторая бинарная операция

```
(pack op ...)          <=> (a[0] op (... op (a[n-2] op a[n-1])))  
(... op pack)          <=> (((a[0] op a[1]) op ...) op a[n-1])  
(pack op ... op init) <=> (a[0] op (... op (a[n-1] op init)))  
(init op ... op pack) <=> (((init op a[0]) op ...) op a[n-1])
```

Fold expression (C++17)

```
(pack op ...)          <=> (a[0] op (... op (a[n-2] op a[n-1])))  
(... op pack)          <=> (((a[0] op a[1]) op ...) op a[n-1])  
(pack op ... op init) <=> (a[0] op (... op (a[n-1] op init)))  
(init op ... op pack) <=> (((init op a[0]) op ...) op a[n-1])
```

Примеры:

```
template <class... Args>  
void Print(std::ostream& os, const Args&... args) {  
    (os << ... << args); // os << args[0] << args[1] ... << args[n-1];  
}
```

```
template <class... Args>  
auto Sum(const Args&... args) {  
    return (args + ...); // args[0] + args[1] + ... + args[n-1]  
    // или return ... + args;  
}
```

Бонус: проблема 1

```
template <class... Args>
void Print(std::ostream& os, const Args&... args) {
    (os << ... << args); // os << args[0] << args[1] ... << args[n-1];
}

Print(std::cout, "a", 1, "b", 2);
```

a1b2

Где пробелы?

Бонус: проблема 1

```
template <class Head, class... Args>
void Print(std::ostream& os, const Head& head, const Args&... args) {
    os << head;
    ((os << ' ' << args), ...); // свертка с операцией "запятая"
    // <=> (os << ' ' << args[0]), ..., (os << ' ' << args[n-1])
}

Print(std::cout, "a", 1, "b", 2);
```

a 1 b 2

Другое дело

Бонус: проблема 2

А если хочется принять произвольное число целых чисел?

```
int SumInt(int... args); // увы, так нельзя  
  
SumInt(1, 2, 3);
```

Бонус: проблема 2

Воспользуемся черной магией

```
template <class... Args,  
          class = std::enable_if_t<(std::is_same_v<Args, int> && ...) >>  
int SumInt(Args... args) {  
    return (args + ...);  
}  
  
SumInt(1, 2, 3);  
// SumInt(1, 2, 3.0);  CE
```

ИЛИ

```
template <class... Args>  
int SumInt(Args... args) {  
    return (static_cast<int>(args) + ...);  
}  
  
SumInt(1, 2, 3);  
SumInt(1, 2, 3.0);  // Ok
```