

`std::initializer_list`

—

~~былинный~~ отказ проектирования

Павел Новиков

 @crr\_are

Align Technology R&D

align

`std::initializer_list`

— epic fail of design

~~был самый большой отказ проектирования~~

Павел Новиков

 @cpr\_are

Align Technology R&D

align

# О чём поговорим

- что такое `std::initializer_list`
- вывод типов аргументов шаблонов и `auto`
- инициализация
- перегрузка функций
- что делать, как жить...

`std::initializer_list`  
не нужен

# Мотивация для `std::initializer_list`

```
std::vector<int> v;
```

```
v.push_back(1);
```

```
v.push_back(2);
```

```
v.push_back(3);
```

```
v.push_back(32);
```

```
v.push_back(42);
```

```
int a[] = { 1, 2, 3, 32, 42 };
```

# Мотивация для `std::initializer_list`

```
std::vector<int> v;
```

```
v.push_back(1);
```

```
v.push_back(2);
```

```
v.push_back(3);
```

```
v.push_back(32);
```

```
v.push_back(42);
```

```
int a[] = { 1, 2, 3, 32, 42 };
```

```
std::vector<int> v = { 1, 2, 3, 32, 42 }; // В C++11
```

# Что такое `std::initializer_list`

Defined in header `<initializer_list>`

```
template< class T >           (since C++11)
class initializer_list;
```

An object of type `std::initializer_list<T>` is a lightweight proxy object that provides access to an array of objects of type `const T`.

A `std::initializer_list` object is automatically constructed when:

- a *braced-init-list* is used to **list-initialize** an object, where the corresponding constructor accepts an `std::initializer_list` parameter
- a *braced-init-list* is used as the right operand of **assignment** or as a **function call argument**, and the corresponding assignment operator/function accepts an `std::initializer_list` parameter
- a *braced-init-list* is bound to **auto**, including in a **ranged for loop**

Initializer lists may be implemented as a pair of pointers or pointer and length. Copying a `std::initializer_list` does not copy the underlying objects.

The underlying array is not guaranteed to exist after the lifetime of the original initializer list object has ended. The storage for `std::initializer_list` is unspecified (i.e. it could be automatic, temporary, or static read-only memory, depending on the situation). (until C++14)

The underlying array is a **temporary** array of type `const T[N]`, in which each element is **copy-initialized** (except that narrowing conversions are invalid) from the corresponding element of the original initializer list. The lifetime of the underlying array is the same as any other **temporary object**, except that initializing an `initializer_list` object from the array extends the lifetime of the array exactly like **binding a reference to a temporary** (with the same exceptions, such as for initializing a non-static class member). The underlying array may be allocated in read-only memory. (since C++14)

The program is ill-formed if an explicit or partial specialization of `std::initializer_list` is declared. (since C++17)

# Что такое `std::initializer_list`

Defined in header `<initializer_list>`

```
template< class T >           (since C++11)
class initializer_list;
```

An object of type `std::initializer_list<T>` is a lightweight proxy object that provides access to an array of objects of type `const T`.

An object of type `std::initializer_list<T>` is a lightweight proxy object that provides access to an array of objects of type `const T`.

- a *braced-init-list* is used as the right operand of `assignment` or as a `function call argument`, and the corresponding assignment operator/function accepts an `std::initializer_list` parameter
- a *braced-init-list* is bound to `auto`, including in a `ranged for loop`

Initializer lists may be implemented as a pair of pointers or pointer and length. Copying a `std::initializer_list` does not copy the underlying objects.

The underlying array is not guaranteed to exist after the lifetime of the original initializer list object has ended. The storage for `std::initializer_list` is unspecified (i.e. it could be automatic, temporary, or static read-only memory, depending on the situation). (until C++14)

The underlying array is a `temporary` array of type `const T[N]`, in which each element is `copy-initialized` (except that narrowing conversions are invalid) from the corresponding element of the original initializer list. The lifetime of the underlying array is the same as any other `temporary object`, except that initializing an `initializer_list` object from the array extends the lifetime of the array exactly like `binding a reference to a temporary` (with the same exceptions, such as for initializing a non-static class member). The underlying array may be allocated in read-only memory. (since C++14)

The program is ill-formed if an explicit or partial specialization of `std::initializer_list` is declared. (since C++17)



# Массив **КОНСТАНТНЫХ** объектов

Member types	
Member type	Definition
value_type	T
reference	const T&
const_reference	const T&
size_type	std::size_t
iterator	const T*
const_iterator	const T*

```
const T* begin() const noexcept;           (since C++11)  
                                           (until C++14)  
constexpr const T* begin() const noexcept; (since C++14)
```

```
const T* end() const noexcept;           (since C++11)  
                                           (until C++14)  
constexpr const T* end() const noexcept; (since C++14)
```

# Массив **КОНСТАНТНЫХ** объектов

```
Foo widget;  
auto v = std::vector<Foo>{ Foo{}, std::move(widget) };
```

ЭКВИВАЛЕНТНО:

```
std::initializer_list<Foo> list{ Foo{}, std::move(widget) };  
auto v = std::vector<Foo>{ list };
```

# Массив **КОНСТАНТНЫХ** объектов

```
Foo widget;
```

```
auto v = std::vector<Foo>{ Foo{}, std::move(widget) };
```

ЭКВИВАЛЕНТНО:

```
std::initializer_list<Foo> list{ Foo{}, std::move(widget) };  
auto v = std::vector<Foo>{ list };
```

только копирование



# Массив **КОНСТАНТНЫХ** объектов

```
std::vector<std::vector<float>> getCoefficients() {  
    //...  
    return { k0, k1, k2 };  
}
```

# Массив **КОНСТАНТНЫХ** объектов

```
std::vector<std::vector<float>> getCoefficients() {  
    //...  
    return { k0, k1, k2 };  
}
```

```
std::vector<Matrix> getLayers() {  
    //...  
    return { a, b, c };  
}
```

# Массив **КОНСТАНТНЫХ** объектов

```
std::vector<std::vector<float>> getCoefficients() {  
    //...  
    return { k0, k1, k2 };  
}
```

копирование 🤪

```
std::vector<Matrix> getLayers() {  
    //...  
    return { a, b, c };  
}
```

# Вывод типов аргументов шаблонов и `auto`

```
foo(42); // T = int  
auto i = 42; // int
```

```
template<typename T>  
void foo(T) {}
```

# Вывод типов аргументов шаблонов и auto

```
foo(42); // T = int
```

```
auto i = 42; // int
```

```
foo(std::string{});
```

```
auto string = std::string{};
```

```
template<typename T>  
void foo(T) {}
```



# Вывод типов аргументов шаблонов и auto

```
foo(42); // T = int
```

```
auto i = 42; // int
```

```
foo(std::string{}); // T = std::string
```

```
auto string = std::string{}; // std::string
```

```
template<typename T>  
void foo(T) {}
```

# Вывод типов аргументов шаблонов и auto

```
foo(42); // T = int
```

```
auto i = 42; // int
```

```
foo(std::string{}); // T = std::string
```

```
auto string = std::string{}; // std::string
```

```
foo("literal");
```

```
auto literal = "literal";
```

```
template<typename T>  
void foo(T) {}
```

# Вывод типов аргументов шаблонов и auto

```
foo(42); // T = int
```

```
auto i = 42; // int
```

```
foo(std::string{}); // T = std::string
```

```
auto string = std::string{}; // std::string
```

```
foo("literal"); // T = const char*
```

```
auto literal = "literal"; // const char*
```

```
template<typename T>  
void foo(T) {}
```

# Вывод типов аргументов шаблонов и `auto`

```
foo({ 1, 2, 3 }); // ?
```

```
template<typename T>  
void foo(T) {}
```

```
auto list = { 1, 2, 3 }; // ?
```

# Вывод типов аргументов шаблонов и `auto`

```
foo({ 1, 2, 3 }); // ?
```

```
template<typename T>  
void foo(T) {}
```

```
auto list = { 1, 2, 3 }; // std::initializer_list<int>
```

# Вывод типов аргументов шаблонов и auto

```
foo({ 1, 2, 3 }); // ?
```

 не скомпилируется

```
template<typename T>  
void foo(T) {}
```

```
auto list = { 1, 2, 3 }; // std::initializer_list<int>
```

- a *braced-init-list* is used to *list-initialize* an object, where the corresponding constructor accepts an `std::initializer_list` parameter
- a *braced-init-list* is used as the right operand of *assignment* or as a *function call argument*, and the corresponding assignment operator/function accepts an `std::initializer_list` parameter

- a *braced-init-list* is used as the right operand of assignment or as a function call argument, and the corresponding assignment operator/function accepts an `std::initializer_list` parameter

static read-only memory, depending on the situation).

The underlying array is a *temporary* array of type `const T[N]`, in which each element is *copy-initialized* (except that narrowing conversions are invalid) from the corresponding element of the original initializer list. The lifetime of the underlying array is the same as any other *temporary object*, except that initializing an `initializer_list` object from the array extends the lifetime of the array exactly like *binding a reference to a temporary* (with the same exceptions, such as for initializing a non-static class member). (since C++14)  
The underlying array may be allocated in read-only memory.

The program is ill-formed if an explicit or partial specialization of `std::initializer_list` is declared. (since C++17)

# Вывод типов аргументов шаблонов и `auto`

```
template<typename T>  
void bar(std::initializer_list<T>) {}  
  
bar({ 1, 2, 3 }); // std::initializer_list<int>
```

# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```



# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```

```
int i4{ 1, 2, 3 };
```

# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3);
```

# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3); // слишком много параметров
```

# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3); // слишком много параметров
```

```
int i6 = { 1, 2, 3 };
```

# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3); // слишком много параметров
```

```
int i6 = { 1, 2, 3 }; // слишком много параметров
```

# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3); // слишком много параметров
```

```
int i6 = { 1, 2, 3 }; // слишком много параметров
```

```
int i7 = (1, 2, 3);
```

# Инициализация

```
int i0{ 42 };
```

```
int i1(42);
```

```
int i2 = { 42 };
```

```
int i3 = (42);
```

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3); // слишком много параметров
```

```
int i6 = { 1, 2, 3 }; // слишком много параметров
```

```
int i7 = (1, 2, 3); // скомпилируется; i7 == 3
```



# Инициализация

MSVC:

error C2440: 'initializing': cannot convert from 'initializer list' to 'int'  
message : The initializer contains too many elements

Clang: error: excess elements in scalar initializer

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3); // слишком много параметров
```

```
int i6 = { 1, 2, 3 }; // слишком много параметров
```

```
int i7 = (1, 2, 3); // скомпилируется; i7 == 3
```

# Инициализация

GCC:

error: scalar object 'i4' requires one element in initializer

error: expression list treated as compound expression in initializer  
[-fpermissive]

error: scalar object 'i6' requires one element in initializer

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3); // слишком много параметров
```

```
int i6 = { 1, 2, 3 }; // слишком много параметров
```

```
int i7 = (1, 2, 3); // скомпилируется; i7 == 3
```

# Инициализация

GCC:

error: scalar object 'i4' requires one element in initializer

error: expression list treated as compound expression in initializer  
[-fpermissive]

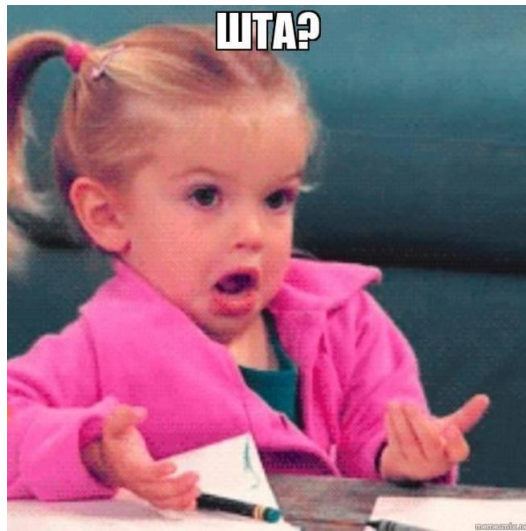
error: scalar object 'i6' requires one element in initializer

```
int i4{ 1, 2, 3 }; // слишком много параметров
```

```
int i5(1, 2, 3); // слишком много параметров
```

```
int i6 = { 1, 2, 3 }; // слишком много параметров
```

```
int i7 = (1, 2, 3); // не работает; i7 == 3
```



# Инициализация

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

`Widget w0{ 42 };` ← вызовется `Widget(int)`

`Widget w1(42);`

`Widget w2{ 42, 23 };` ← вызовется `Widget(int, int)`

`Widget w3(42, 23);`

# Инициализация

```
template <class T, class A = std::allocator<T>>
class std::vector {
    explicit vector(const size_type count, const A& = A());
    vector(const size_type count, const T, const A& = A());
    vector(std::initializer_list<T>, const A& = A());
};
```

```
std::vector<char> v0{ 32 };
```

```
std::vector<char> v1(32);
```

```
std::vector<char> v2{ 32 , 42 };
```

```
std::vector<char> v3(32, 42);
```

# Инициализация

```
template <class T, class A = std::allocator<T>>
class std::vector {
    explicit vector(const size_type count, const A& = A());
    vector(const size_type count, const T, const A& = A());
    vector(std::initializer_list<T>, const A& = A());
};

std::vector<char> v0{ 32 }; // { ' ' }
std::vector<char> v1(32); // { '\0', '\0', ... } '\0'x32
std::vector<char> v2{ 32 , 42 }; // { ' ', '*' }
std::vector<char> v3(32, 42); // { '*', '*', ... } '*'x32
```

# Инициализация

## `std::vector<T,Allocator>::emplace_back`

```
template< class... Args >                               (since C++11)  
void emplace_back( Args&&... args );                     (until C++17)  
  
template< class... Args >                               (since C++17)  
reference emplace_back( Args&&... args );
```

Appends a new element to the end of the container. The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at the location provided by the container. The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`.

If the new `size()` is greater than `capacity()` then all iterators and references (including the past-the-end iterator) are invalidated. Otherwise only the past-the-end iterator is invalidated.

The element is constructed through `std::allocator_traits::construct`

# Инициализация

`std::vector<T,Allocator>::emplace_back`

If the above is not possible (e.g. a does not have the member function `construct()`), then calls placement-new as

```
::new (static_cast<void*>(p)) T(std::forward<Args>(args)...) )
```

Appends a new element to the end of the container. The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at the location provided by the container. The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`.

If `ar`  
`std::allocator_traits<Alloc>::construct`

Defined in header `<memory>`

```
template< class T, class... Args >
static void construct( Alloc& a, T* p, Args&&... args ); (since C++11)
```

If possible, constructs an object of type `T` in allocated uninitialized storage pointed to by `p`, by calling

```
a.construct(p, std::forward<Args>(args)...) )
```

If the above is not possible (e.g. `a` does not have the member function `construct()`), then calls placement-new as

```
::new (static_cast<void*>(p)) T(std::forward<Args>(args)...) )
```



# Инициализация

```
struct Bar {  
    int i;  
};  
std::vector<Bar> gadgets;  
gadgets.emplace_back(42); // зовёт Bar(42)
```

 не скомпилируется

# Инициализация

```
template<typename T>
struct CurlyBracesAllocator {
    using value_type = T;
    static constexpr std::align_val_t alignment =
        static_cast<std::align_val_t>(alignof(T));
    [[nodiscard]] constexpr T *allocate(std::size_t n) {
        return static_cast<T *> (::operator new(n, alignment));
    }
    constexpr void deallocate(T *p, std::size_t n) noexcept {
        ::operator delete(static_cast<void *>(p), n, alignment);
    }
    template<typename U, typename... Args>
    static void construct(U *p, Args&&... args) {
        ::new(static_cast<void *>(p)) U{ std::forward<Args>(args)... };
    }
};
```

# Инициализация

```
template<typename T>
struct CurlyBracesAllocator {
    using value_type = T;
    static constexpr std::align_val_t alignment =
        static_cast<std::align_val_t>(alignof(T));
    [[nodiscard]] constexpr T *allocate(std::size_t n) {
        return static_cast<T *> (::operator new(n, alignment));
    }
    constexpr void deallocate(T *p, std::size_t n) noexcept {
        ::operator delete(static_cast<void *>(p), n, alignment);
    }
    template<typename U, typename... Args>
    static void construct(U *p, Args&&... args) {
        ::new(static_cast<void *>(p)) U{ std::forward<Args>(args)... };
    }
};
```

# Инициализация

```
template<typename T>
struct CurlyBracesAllocator {
    //...
    template<typename U, typename... Args>
    static void construct(U *p, Args&&... args) {
        ::new(static_cast<void*>(p)) U{ std::forward<Args>(args)... };
    }
};
std::vector<Gadget, CurlyBracesAllocator<Gadget>> gadgets;
gadgets.emplace_back(42);
```

# Инициализация

## aggregate initialization

Initializes an aggregate from braced-init-list

### Syntax

<code>T object = {arg1, arg2, ...};</code>	(1)	
<code>T object {arg1, arg2, ...};</code>	(2)	(since C++11)
<code>T object = { .designator = arg1 , .designator { arg2 } ... };</code>	(3)	(since C++20)
<code>T object { .designator = arg1 , .designator { arg2 } ... };</code>	(4)	(since C++20)
<code>T object (arg1, arg2, ...);</code>	(5)	(since C++20)

```
std::vector<Bar> gadgets;
```

```
gadgets.emplace_back(42); // должно работать в C++20
```

# Перегрузка функций

```
void baz(const Widget&) {}
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

```
baz(Widget{ 4, 8 });
```

# Перегрузка функций

```
void baz(const Widget&) {}
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

```
baz(Widget{ 4, 8 });
```

```
baz({ 15, 16 });
```

# Перегрузка функций

```
void baz(const Widget&) {}
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

```
baz(Widget{ 4, 8 });
```

```
baz({ 15, 16 }); // зовёт baz(Widget)
```



**ПРИВЕТ, C++ СИБИРЬ**

**Я ХОЧУ СЫГРАТЬ С ТОБОЙ В ОДНУ ИГРУ...**

# Перегрузка функций

```
void baz(const Widget&) {}
```

```
void baz(std::initializer_list<int>) {}
```

```
baz(Widget{ 4, 8 });
```

```
baz({ 15, 16 });
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

# Перегрузка функций

```
void baz(const Widget&) {}
```

```
void baz(std::initializer_list<int>) {}
```

```
baz(Widget{ 4, 8 });
```

```
baz({ 15, 16 }); // зовёт baz(std::initializer_list<int>)
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

# Перегрузка функций

```
void baz(const Widget&) {}
```

```
namespace evil {
```

```
void baz(std::initializer_list<int>) {}
```

```
}
```

```
baz(Widget{ 4, 8 });
```

```
baz({ 15, 16 });
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

# Перегрузка функций

```
void baz(const Widget&) {}
```

```
namespace evil {
```

```
void baz(std::initializer_list<int>) {}
```

```
}
```

```
baz(Widget{ 4, 8 });
```

```
baz({ 15, 16 }); // зовёт baz(const Widget&)
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

# Перегрузка функций

```
void baz(const Widget&) {}  
namespace evil {  
void baz(std::initializer_list<int>) {}  
}  
using namespace evil;  
baz(Widget{ 4, 8 });  
baz({ 15, 16 });
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

# Перегрузка функций

```
void baz(const Widget&) {}  
namespace evil {  
void baz(std::initializer_list<int>) {}  
}  
using namespace evil;  
baz(Widget{ 4, 8 });  
baz({ 15, 16 }); // зовёт baz(std::initializer_list<int>)
```

```
struct Widget {  
    Widget(int) {}  
    Widget(int, int) {}  
};
```

# std::initializer\_list бесполезен?

```
for (auto &w : { Widget{42}, Widget{23} })  
    baz(w);
```



# std::initializer\_list бесполезен?

```
for (auto &w : { Widget{42}, Widget{23} })
    baz(w);

{
    auto &&range = { Widget{42}, Widget{23} };
    auto begin = range.begin();
    auto end = range.end();
    for (; begin != end; ++begin) {
        auto &w = *begin;
        baz(w);
    }
}
```

# std::initializer\_list бесполезен?

```
for (auto &w : { Widget{42}, Widget{23} })  
    baz(w);
```

std::initializer\_list<Widget>

```
{  
    auto &&range = { Widget{42}, Widget{23} };  
    auto begin = range.begin();  
    auto end = range.end();  
    for (; begin != end; ++begin) {  
        auto &w = *begin;  
        baz(w);  
    }  
}
```



# Что делать, как жить

Чего бы нам хотелось?

- легко писать код правильно, сложно неправильно;  
без сюрпризов,  
ошибка компиляции при неопределённости

# Что делать, как жить

Чего бы нам хотелось?

- легко писать код правильно, сложно неправильно;  
без сюрпризов,  
ошибка компиляции при неопределённости

```
std::vector<char> v2{ 42 , 23 };
```

```
std::vector<char> v3(42, 23);
```



# Что делать, как жить

Чего бы нам хотелось?

- легко писать код правильно, сложно неправильно;  
без сюрпризов,  
ошибка компиляции при неопределённости

```
std::vector<char> v2{ 42 , 23 };
```

```
std::vector<char> v3(42, 23);
```

- легко понять написанный код



# Что делать, как жить

Чего бы нам хотелось?

- легко писать код правильно, сложно неправильно;  
без сюрпризов,  
ошибка компиляции при неопределённости

```
std::vector<char> v2{ 42 , 23 };
```

```
std::vector<char> v3(42, 23);
```

- легко понять написанный код
- эффективность?



Если ты в чистом поле...

```
struct Baz {  
    template<typename... T>  
    Baz(T&&... args) { (std::forward<T>(args), ...); }  
};
```

Если ты в чистом поле...

```
struct Baz {  
    template<typename... T>  
    Baz(T&&... args) { (std::forward<T>(args), ...); }  
};  
void qux(Baz) {}  
void qux(std::vector<const char*>) {}  
  
qux({ "hello", "there" }); // неоднозначный вызов  
                             // перегруженной функции
```



Если ты в чистом поле...

```
struct Baz {  
    template<typename... T,  
            typename =  
                std::enable_if_t<  
                    (std::is_convertible_v<T, int>&&...)  
                >  
    >  
    Baz(T&&... args) { (std::forward<T>(args), ...); }  
};
```

Если ты в чистом поле...

```
struct Baz {  
    template<typename... T,  
            typename =  
                std::enable_if_t<  
                    (std::is_convertible_v<T, int>&&...)  
                >  
            > requires (std::is_convertible_v<T, int>&&...)  
    Baz(T&&... args) { (std::forward<T>(args), ...); }  
};
```

Если ты в чистом поле...

```
struct Baz {  
    template<typename... T>  
    requires (std::is_convertible_v<T, int>&&...)  
    Baz(T&&... args) { (std::forward<T>(args), ...); }  
};  
void qux(Baz) {}  
void qux(std::vector<const char*>) {}  
  
qux({ "hello", "there" }); // qux(vector<const char*>)
```

# Есть перегруженные функции/конструкторы

```
struct Gadget {  
    Gadget(int) {}  
    Gadget(int, char) {}  
    template<typename... U>  
    Gadget(U&&... args) {  
        (std::forward<U>(args), ...);  
    }  
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (U&&...)
Gadget g3( 32, 42 ); // (U&&...)
Gadget g4{ 1, 2, 3 }; // (U&&...)
Gadget g5( 1, 2, 3 ); // (U&&...)
```

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<typename... U>
    Gadget(U&&...) {}
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (U&&...)
Gadget g3( 32, 42 ); // (U&&...)
Gadget g4{ 1, 2, 3 }; // (U&&...)
Gadget g5( 1, 2, 3 ); // (U&&...)
```

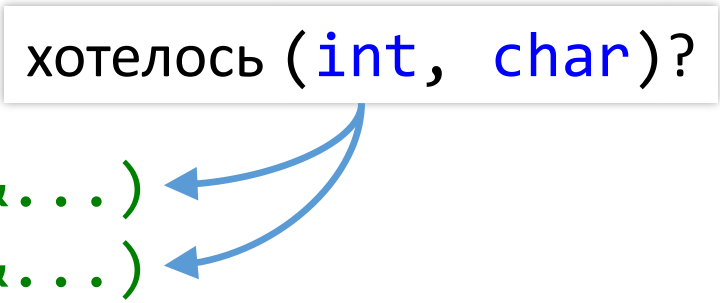
хотелось (int, char)?

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<typename... U>
    Gadget(U&&...) {}
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (U&&...)
Gadget g3( 32, 42 ); // (U&&...)
Gadget g4{ 1, 2, 3 }; // (U&&...)
Gadget g5( 1, 2, 3 ); // (U&&...)
Gadget g6{{ 1, 2, 3 }}; // нет перегрузки для std::initializer_list
```

хотелось (int, char)?



```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<typename... U>
    Gadget(U&&...) {}
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (U&&...)
Gadget g3( 32, 42 ); // (U&&...)
Gadget g4{ 1, 2, 3 }; // (U&&...)
Gadget g5( 1, 2, 3 ); // (U&&...)
Gadget g6{{ 1, 2, 3 }}; // нет перегрузки для std::initializer_list
Gadget g7({ 1, 2, 3 }); // (U&&...)
```

хотелось (int, char)?

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<typename... U>
    Gadget(U&&...) {}
};
```



# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (U&&...)
Gadget g3( 32, 42 ); // (U&&...)
Gadget g4{ 1, 2, 3 }; // (U&&...)
Gadget g5( 1, 2, 3 ); // (U&&...)
Gadget g6{{ 1, 2, 3 }}; // нет перегрузки для std::initializer_list
Gadget g7({ 1, 2, 3 }); // (U&&...)
Gadget g8{( 1, 2, 3 )}; // (int)
Gadget g9(( 1, 2, 3 )); // (int)
Gadget g10{{( 1, 2, 3 )}}; // (int)
Gadget g11({( 1, 2, 3 )}); // (int)
```

хотелось (int, char)?

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<typename... U>
    Gadget(U&&...) {}
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (U&&...)
Gadget g3( 32, 42 ); // (U&&...)
Gadget g4{ 1, 2, 3 }; // (U&&...)
Gadget g5( 1, 2, 3 ); // (U&&...)
Gadget g6{{ 1, 2, 3 }}; // нет перегрузки для std::initializer_list
Gadget g7({ 1, 2, 3 }); // (U&&...)
Gadget g8{( 1, 2, 3 )}; // (int)
Gadget g9(( 1, 2, 3 )); // (int)
Gadget g10{{( 1, 2, 3 )}}; // (int)
Gadget g11({( 1, 2, 3 )}); // (int)
Gadget g12{({ 1, 2, 3 })}; // синтаксическая ошибка
Gadget g13(({ 1, 2, 3 })); // синтаксическая ошибка
```

хотелось (int, char)?

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<typename... U>
    Gadget(U&&...) {}
};
```

# Есть перегруженные функции/конструкторы

```
struct Gadget {  
    Gadget(int) {}  
    Gadget(int, char) {}  
    template<size_t N>  
    Gadget(char(&&a)[N]) {  
        for (auto &i : a)  
            std::move(i);  
    }  
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (int, char)
Gadget g3( 32, 42 ); // (int, char)
```

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<size_t N>
    Gadget(char(&&a)[N]) {}
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (int, char)
Gadget g3( 32, 42 ); // (int, char)
Gadget g4{ 1, 2, 3 }; // нет перегрузки для std::initializer_list
Gadget g5( 1, 2, 3 ); // нет перегрузки для трёх аргументов
```

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<size_t N>
    Gadget(char(&&a)[N]) {}
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (int, char)
Gadget g3( 32, 42 ); // (int, char)
Gadget g4{ 1, 2, 3 }; // нет перегрузки для std::initializer_list
Gadget g5( 1, 2, 3 ); // нет перегрузки для трёх аргументов
Gadget g6{{ 1, 2, 3 }}; // (int(&&)[3])
Gadget g7({ 1, 2, 3 }); // (int(&&)[3])
```

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<size_t N>
    Gadget(char(&&a)[N]) {}
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (int, char)
Gadget g3( 32, 42 ); // (int, char)
Gadget g4{ 1, 2, 3 }; // нет перегрузки для std::initializer_list
Gadget g5( 1, 2, 3 ); // нет перегрузки для трёх аргументов
Gadget g6{{ 1, 2, 3 }}; // (int(&&)[3])
Gadget g7({ 1, 2, 3 }); // (int(&&)[3])
Gadget g8{( 1, 2, 3 )}; // (int)
Gadget g9(( 1, 2, 3 )); // (int)
Gadget g10{{( 1, 2, 3 )}}; // (int)
Gadget g11({( 1, 2, 3 )}); // (int)
```

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<size_t N>
    Gadget(char(&&a)[N]) {}
};
```

# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (int, char)
Gadget g3( 32, 42 ); // (int, char)
Gadget g4{ 1, 2, 3 }; // нет перегрузки для std::initializer_list
Gadget g5( 1, 2, 3 ); // нет перегрузки для трёх аргументов
Gadget g6{{ 1, 2, 3 }}; // (int(&&)[3])
Gadget g7({ 1, 2, 3 }); // (int(&&)[3])
Gadget g8{( 1, 2, 3 )}; // (int)
Gadget g9(( 1, 2, 3 )); // (int)
Gadget g10{{( 1, 2, 3 )}}; // (int)
Gadget g11({( 1, 2, 3 )}); // (int)
Gadget g12{({ 1, 2, 3 })}; // синтаксическая ошибка
Gadget g13(({ 1, 2, 3 })); // синтаксическая ошибка
```

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<size_t N>
    Gadget(char(&&a)[N]) {}
};
```



# Есть перегруженные функции/конструкторы

```
Gadget g0{ 32 }; // (int)
Gadget g1( 32 ); // (int)
Gadget g2{ 32, 42 }; // (int, char)
Gadget g3( 32, 42 ); // (int, char)
Gadget g4{ 1, 2, 3 }; // нет перегрузки для std::initializer_list
Gadget g5( 1, 2, 3 ); // нет перегрузки для трёх аргументов
Gadget g6{{ 1, 2, 3 }}; // (int(&&)[3])
Gadget g7({ 1, 2, 3 }); // (int(&&)[3])
Gadget g8{( 1, 2, 3 )}; // (int)
Gadget g9(( 1, 2, 3 )); // (int)
Gadget g10{{( 1, 2, 3 )}}; // (int)
Gadget g11({( 1, 2, 3 )}); // (int)
Gadget g12{({ 1, 2, 3 })}; // синтаксическая ошибка
Gadget g13(({ 1, 2, 3 })); // синтаксическая ошибка
```

```
struct Gadget {
    Gadget(int) {}
    Gadget(int, char) {}
    template<size_t N>
    Gadget(char(&&a)[N]) {}
};
```

Есть перегруженные функции/конструкторы

```
void quux(Gadget) {}
```

```
void quux(std::vector<const char*>) {}
```

```
quux(Gadget{ 32, 42 });
```

```
quux(Gadget{ {32, 42} });
```

Есть перегруженные функции/конструкторы

```
void quux(Gadget) {}
```

```
void quux(std::vector<const char*>) {}
```

```
quux(Gadget{ 32, 42 }); // Gadget(int, char)
```

```
quux(Gadget{ {32, 42} }); // Gadget(char(&&)[2])
```

Есть перегруженные функции/конструкторы

```
void quux(Gadget) {}
```

```
void quux(std::vector<const char*>) {}
```

```
quux(Gadget{ 32, 42 }); // Gadget(int, char)
```

```
quux(Gadget{ {32, 42} }); // Gadget(char(&&)[2])
```

```
quux({ 32, 42 });
```

```
quux({ {32, 42} });
```

# Есть перегруженные функции/конструкторы

```
void quux(Gadget) {}
```

```
void quux(std::vector<const char*>) {}
```

```
quux(Gadget{ 32, 42 }); // Gadget(int, char)
```

```
quux(Gadget{ {32, 42} }); // Gadget(char(&&)[2])
```

```
quux({ 32, 42 }); // Gadget(int, char)
```

```
quux({ {32, 42} });
```

# Есть перегруженные функции/конструкторы

```
void quux(Gadget) {}
```

```
void quux(std::vector<const char*>) {}
```

```
quux(Gadget{ 32, 42 }); // Gadget(int, char)
```

```
quux(Gadget{ {32, 42} }); // Gadget(char(&&)[2])
```

```
quux({ 32, 42 }); // Gadget(int, char)
```

```
quux({ {32, 42} }); // Gadget(char(&&)[2])
```

# Есть перегруженные функции/конструкторы

```
void quux(Gadget) {}
```

```
void quux(std::vector<const char*>) {}
```

```
quux(Gadget{ 32, 42 }); // Gadget(int, char)
```

```
quux(Gadget{ {32, 42} }); // Gadget(char(&&)[2])
```

```
quux({ 32, 42 }); // Gadget(int, char)
```

```
quux({ {32, 42} }); // Gadget(char(&&)[2])
```

```
quux({ { "general", "kenobi" } });
```

# Есть перегруженные функции/конструкторы

```
void quux(Gadget) {}
```

```
void quux(std::vector<const char*>) {}
```

```
quux(Gadget{ 32, 42 }); // Gadget(int, char)
```

```
quux(Gadget{ {32, 42} }); // Gadget(char(&&)[2])
```

```
quux({ 32, 42 }); // Gadget(int, char)
```

```
quux({ {32, 42} }); // Gadget(char(&&)[2])
```

```
quux({ { "general", "kenobi" } }); // vector<const char*>{}
```



# Выводы?

# Выводы?

- Проведите своё собственное исследование.  
Никто не знает ваш код лучше, чем вы, поэтому никто не скажет что для вас лучше, чем вы сами.

# Выводы?

- Проведите своё собственное исследование.  
Никто не знает ваш код лучше, чем вы, поэтому никто не скажет что для вас лучше, чем вы сами.
- Не используйте `std::initializer_list...`  
по крайней мере пока его не исправят.

# Выводы?

- Проведите своё собственное исследование.  
Никто не знает ваш код лучше, чем вы, поэтому никто не скажет что для вас лучше, чем вы сами.
- Не используйте `std::initializer_list...`  
по крайней мере пока его не исправят.
- В C++ много средств для выразительного написания эффективного кода.  
Изучите существующие средства,  
придумайте свои, если существующих недостаточно.

`std::initializer_list` — былинный отказ проектирования

Павел Новиков

 @cpr\_are

Align Technology R&D

align