

Язык C++

Variadic template.

to_string

```
template<typename T>
std::string to_string( const T& value) {
    std::stringstream ss;
    ss << value;
    return ss.str();
}
```

Variadic template (C++ 11)

```
template<typename... TArgs>  
std::vector<std::string> to_strings(const TArgs&... args);
```

Variadic template

```
std::vector<std::string> to_strings() {  
    return {} ;  
}  
  
template<typename T, typename... TArgs>  
std::vector<std::string> to_strings(const T& value, const TArgs&... args) {  
    std::vector<std::string> result;  
    result.push_back(to_string(value));  
  
    std::vector<std::string> other = to_strings(args...);  
    result.insert(result.end(), other.begin(), other.end());  
    return result;  
}
```

Variadic templates

- parameter pack
- const/volatile
- sizeof...
- fold-expression

Parameter pack

```
template<typename T, typename... TArgs>
void printAll(const T& v, const TArgs&... args) {
    std::cout << v << " ";

    if constexpr(sizeof...(args) > 0) {
        printAll(args...);
    }
}
```

- A function parameter pack with an optional name
- A type template parameter pack with an optional name
- sizeof... operator - queries the number of elements in a parameter pack
- Parameter pack expansion

Fold-expression (C++ 17)

```
template<typename... TArgs>
std::vector<std::string> to_strings(const TArgs&... args) {
    return {to_string(args)...};
}
```

Fold-expression (C++ 17)

- Unary right fold ($E \text{ op } \dots$) becomes $(E_1 \text{ op } (\dots \text{ op } (E_{N-1} \text{ op } E_N)))$
- Unary left fold ($\dots \text{ op } E$) becomes $((E_1 \text{ op } E_2) \text{ op } \dots) \text{ op } E_N$
- Binary right fold ($E \text{ op } \dots \text{ op } I$) becomes $(E_1 \text{ op } (\dots \text{ op } (E_{N-1} \text{ op } (E_N \text{ op } I))))$
- Binary left fold ($I \text{ op } \dots \text{ op } E$) becomes $((((I \text{ op } E_1) \text{ op } E_2) \text{ op } \dots) \text{ op } E_N)$

Fold-expression (C++ 17)

```
template<typename ...TArgs>
auto multiply(TArgs... args)  {
    return (args * ... );
}

template<typename ...TArgs>
auto divide(TArgs... args)  {
    return (args / ... );
}
```

Fold-expression (C++ 17)

```
template<typename ...TArgs>
auto divide(TArgs... args)  {
    return ( ... / args );
}

template<typename ...TArgs>
auto divide(TArgs... args)  {
    return (1.0 / ... / args );
}
```

Tuple

```
template<typename... TValue>
struct NaiveTuple;

template<>
struct NaiveTuple<> {
};
```

Tuple

```
template<typename... TValue>
struct NaiveTuple;

template<>
struct NaiveTuple<> {
};
```

```
template<typename Head, typename... Tail>
struct NaiveTuple<Head, Tail...> : NaiveTuple<Tail...>
{
    using Base = NaiveTuple<Tail...>;
    using value_type = Head;

    NaiveTuple(const Head& h, const Tail&... tail)
        : NaiveTuple<Tail...>(tail...)
        , head(h)
    {}

    Base& base = static_cast<Base&>(*this);
    Head head;
};
```

Tuple

```
template<size_t I, typename Head, typename... Tail>
struct tuple_element {
    using ElementType = typename tuple_element<I-1, Tail...>::ElementType;

    static ElementType get(const NaiveTuple<Head, Tail...>& t) {
        return tuple_element<I-1, Tail...>::get(t);
    }
};
```

Tuple

```
template<typename Head, typename... Tail>
struct tuple_element<0, Head, Tail...> {
    using ElementType = typename NaiveTuple<Head, Tail...>::value_type;

    static ElementType get(const NaiveTuple<Head, Tail...>& t) {
        return t.head;
    }
};
```

Tuple

```
template<size_t I, typename... TArgs>
auto get(const NaiveTuple<TArgs...>& t) {
    return tuple_element<I, TArgs...>::get(t);
}
```

Tuple

```
template<size_t I, typename... TArgs>
auto get(const NaiveTuple<TArgs...>& t) {
    return tuple_element<I, TArgs...>::get(t);
}

template<typename... TArgs>
NaiveTuple<TArgs...> make_tuple(TArgs... args) {
    return Tuple<TArgs...>(args...);
}
```


Deduction guide

- Class Template Argument Deduction (CTAD)
- Нет возможности вывести тип класса если аргументы с ним не связаны

```
std::array<int, 5> arr = {1,2,3,4,5};
std::vector v(arr.begin(), arr.end());

////////// from vector implementation

template<class _InputIterator,
         class _Alloc = allocator<__iter_value_type<_InputIterator>>,
         class = _EnableIf<__is_allocator<_Alloc>::value>
         >
vector(_InputIterator, _InputIterator)
-> vector<__iter_value_type<_InputIterator>, _Alloc>;
```

Lambda (кратко)

- Не именованный функциональный объект
- Может “захватывать” переменные
- `&`, `=`, `this`, `*this`

Lambda (кратко)

```
struct Less5 {  
    bool operator()(int i) {  
        return i < 5;  
    }  
};  
  
int main() {  
    std::vector v = {1,2,3,4,5,6,7};  
  
    std::cout << std::count_if(v.begin(), v.end(), Less5{}) << std::endl;  
    std::cout << std::count_if(v.begin(), v.end(), [](int i){ return i < 5;}) << std::endl;  
    return 0;  
}
```

Lambda (кратко)

```
struct LessN {
    explicit LessN(int max) : max_(max) {}

    bool operator()(int i) {
        return i < max_;
    }
private:
    int max_;
};

int main() {
    std::vector v = {1,2,3,4,5,6,7};
    int max_value = 4;
    std::cout << std::count_if(v.begin(), v.end(), LessN{max_value}) << std::endl;
    std::cout << std::count_if(v.begin(), v.end(), [max_value]int i){ return i < max_value;}) << std::endl;
    return 0;
}
```

Lambda (кратко)

```
struct LessN {  
    explicit LessN(int max) : max_(max) {}  
    bool operator()(int i) {  
        return i < max_;  
    }  
private:  
    int max_;  
};  
  
int main() {  
    std::vector v = {1,2,3,4,5,6,7};  
  
    std::cout << std::count_if(v.begin(), v.end(), LessN{3}) << std::endl;  
    std::cout << std::count_if(v.begin(), v.end(), [max_value=3](int i){ return i < max_value;}) << std::endl;  
  
    return 0;  
}
```

Overload pattern

```
template<typename ... Ts>  
struct Overload : Ts ... {  
    using Ts::operator() ...;  
};
```

```
template<typename... Ts> Overload(Ts...) -> Overload<Ts...>;
```

Overload pattern

```
template<typename T>
struct Foo {
    void operator()(const T& value) {
        std::cout << "Foo::operator() ";
    }
};

int main(int, char**) {
    auto overload = Overload {
        [](char) { std::cout << "char"; },
        [](int) { std::cout << "int"; },
        [](long) { std::cout << "long"; },
        Foo<std::string>{}
    };

    overload(1);
    overload("string");
    overload(true);
}
```

std::variant

- Строго типизированный Union
- Хранит одно из значений из списка
- `valueless_by_exception`
- `std::bad_variant_access`
- `std::get<>`

```
std::variant<int, long, std::string> v = 11;
std::cout << std::get<long>(v) << std::endl;

try {
    std::cout << std::get<int>(v) << std::endl;
} catch (const std::bad_variant_access & e) {
    std::cout << e.what() << std::endl;
}
```


std::visit

```
int main(int, char**) {  
    std::variant<int, long, std::string> v = 11;  
  
    auto overload = Overload {  
        [](char) { std::cout << "char"; },  
        [](int) { std::cout << "int"; },  
        [](long) { std::cout << "long"; },  
        [](const std::string&) { std::cout << "std::string"; }  
    };  
  
    std::visit(overload, v);  
}
```