



Uniwersytet  
Wrocławski

# Szablony o nieokreślonej liczbie parametrów (*variadic templates*)

Zbigniew Koza

Wydział Fizyki i Astronomii

# Ograniczenie szablonów C++98

- Liczba parametrów szablonów musi być znana podczas kompilacji

```
template<typename T0>  
print(T0 t0);
```

```
template<typename T0, typename T1> print(T0 t0,  
T1 t1);
```

```
template<typename T0, typename T1, typename T2>  
print(T0 t0, T1 t1, T2 t2);
```

...

# C++11: *parameter pack...*

```
template <class... Ts>
```

```
template <int... Ns>
```

```
template <template <class T>... class Us>
```

- **Ts** jest listą parametrów określających typy
- **Ns** jest listą parametrów liczbowych (`int`)
- **Us** jest listą parametrów generowanych z szablonów

# C++11: *parameter pack*

```
template <class... Ts>
template <int... Ns>
template <template <class T>... class Us>
```

Lista  
(*plural*)

- **Ts** jest listą parametrów określających typy
- **Ns** jest listą parametrów liczbowych (`int`)
- **Us** jest listą parametrów generowanych z szablonów

# Variadic templates

- Mogą być używane do definiowania szablonów **klas i funkcji**
- Można więc definiować struktury danych o nieokreślonej liczbie i typach składowych i funkcje o nieokreślonej liczbie i typach argumentów
- Mogą podlegać **specjalizacji**
- W definicji szablonu można używać parametrów pojedynczych i zagregowanych

```
template <typename T, int N, typename... Ts>
```



The diagram consists of three arrows originating from the text 'pojedynczych i zagregowanych' in the previous block. A green arrow points to the parameter **T**, another green arrow points to the parameter **N**, and a purple arrow points to the parameter **Ts**.

# *Function parameter packs*

```
template <typename... Args> //Args: template parameter pack
void f(int i, Args... args) //args: function parameter pack
{
    //...
}


int main()
{
    f (2, "Ala", 3.14, main);
}
```

- **Lista** argumentów funkcji rozwijanych z „paczki”

# *Pusty parameter packs*

```
template <typename... Args>
void f(int i, Args... args)
{
    //...
}

int main()
{
    f (2);
}
```



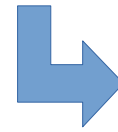
Ten  
przecinek  
będzie  
zignorowany

- Jeżeli lista args jest pusta, kod zostanie skompilowany tak, jak się tego spodziewamy

# Operator **sizeof**...

```
#include <iostream>
template <typename... Args>
void f(Args... args)
{
    std::cout << sizeof...(Args) << "\t";
    std::cout << sizeof...(args) << "\n";
}

int main()
{
    f("Ala", 3.14, main);
    f("Ola");
    f();
}
```



3	3
1	1
0	0

- Zwraca stałą czasu kompilacji



# \_\_PRETTY\_FUNCTION\_\_

```
template <typename... Args>
void f(Args... args)
{
    std::cout << __PRETTY_FUNCTION__ << "\n";
}

int main()
{
    f("Ala", 3.14, main);
    f("Ola");
    f();
}
```



```
void f(Args ...) [with Args = {const char*, double, int (*)(())}]
void f(Args ...) [with Args = {const char*}]
void f(Args ...) [with Args = {}]
```

# Rozwinięcie listy (*pack expansion*)

- **args...** rozwija się do listy argumentów oddzielonych przecinkami
- **expr(args)...** rozwija się do listy **expr(E1), expr(E2),..., expr(EN)**
- **...** zawsze występuje **po** rozwijanym wyrażeniu

# Gdzie można użyć listy argumentów oddzielonych przecinkami?

Argumenty wywołania funkcji  
( $\rightarrow$  quasi rekursja)

$(E1, E2, \dots, EN)$

Lista inicjalizacyjna

$\{E1, E2, \dots, EN\}$

- Wyrażenie z operatorem przecinkowym

$(E1, E2, \dots, EN)$

# Packet expansion – przykład 1

```
template <typename... Args>
void f(Args... args)
{
    auto list = {args...};
    for (auto n : list)
        std::cout << n << " ";
    std::cout << "\n";
}

int main()
{
    f(1, 2, 3);
    f("Ola", "Ała");
}
```

Proste rozwinięcie  
do listy  
inicjalizacyjnej

```
1 2 3
Ola Ała
```

# Przykład 2 - wyrażenie

```
template <typename... Args>
void f(Args... args)
{
    auto list = {(args + 10)...};
    for (auto n : list)
        std::cout << n << " ";
    std::cout << "\n";
}

int main()
{
    f(1, 2, 3);
    f(2.5, 3.14);
}
```

A nawet tak:

```
auto list = {10 + args...};
```



```
11 12 13
12.5 13.14
```

```

template<typename T>
T square(T x)
{
    return x * x;
}

template <typename... Args>
void f(Args... args)
{
    auto list = {square(args)...};
    for (auto n : list)
        std::cout << n << " ";
    std::cout << "\n";
}

int main()
{
    f(1, 2, 3);
    f(2.5, 3.14);
}

```

## Przykład 3



```

1 4 9
6.25 9.8596

```

# To rozwiązanie ma oczywiste wady

- Typy parametrów muszą być takie same
- Nie można użyć pustej listy argumentów

```
f(1, 2, 3);  
f(2.5, 3.14);  
// f(); błąd: brak argumentów  
// f(1, 3.14); błąd: argumenty różnych typów
```

# „Oszukajmy” kompilator

```
template<typename T>
T square(T x) { return x * x; }

template <typename... Args>
void f(Args... args)
{
    auto list = {(square(args), 0)...};
    for (auto n : list)
        std::cout << n << " ";
    std::cout << "\n";
}

int main()
{
    f(1, 2, 3);
    f(2.5, 3.14);
    // f(); błąd: auto list = {} nie ma sensu
    f(1, 3.14, 'c');
}
```

Zawsze  
zero!

0	0	0
0	0	
0	0	0

Operator  
przecinkowy

funkcja  
wywoływana  
na każdym  
argumencie!

Działa z  
argumentami  
różnych typów

```
auto list = {(square(E1), 0), (square(E2), 0), ..., (square(EN), 0)};
```



```

template<typename T>
T square(T x)
{
    std::cout << "squaring " << x << "\n";
    return x * x;
}

template <typename... Args>
void f(Args... args)
{
    (void)std::initializer_list<int> {((void)square(args), 0)...};
}

int main()
{
    f(1, 3.14, 'c');
}

```

→

```

squaring 1
squaring 3.14
squaring c

```

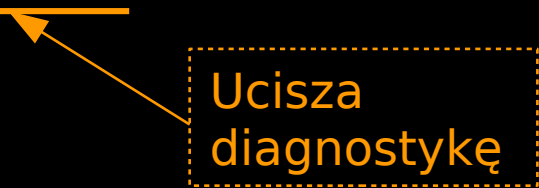
- Wywołanie funkcji: efekt uboczny wywołania operator,
- (void) ucisza diagnostykę kompilatora/analizatora kodu i zabezpiecza przed niespodziankami, jakie może sprawić typ funkcji square (np. przeciążony operator,)
- auto zbędne, bo znamy typ elementów listy (zera są typu int)

# To samo w duchu C++17

```
template<typename T>
T square(T x)
{
    std::cout << "squaring " << x << "\n";
    return x * x;
}

template <typename... Args>
void f(Args... args)
{
    [[maybe_unused]] auto x = {((void)square(args), 0)...};
}

int main()
{
    f(1, 3.14, 'c');
}
```



Ucisza  
diagnostykę

# Przetwarzanie argument po argumencie

```
template <typename Head, typename... Tail>
void print(Head const& head, Tail const&... tail){
    std::cout << head;
    if constexpr(sizeof...(tail) > 0)
    {
        std::cout << ", ";
        print(tail...);
    }
    else
        std::cout << "\n";
}

int main()
{
    print(1, 3.14, 'c', "Ala", std::string("0la"));
}
```

# Przetwarzanie argument po argumencie

```
template <typename Head, typename... Tail>
void print(Head const& head, Tail const&... tail){
    std::cout << head;
    if constexpr(sizeof...(tail) > 0)
    {
        std::cout << ", ";
        print(tail...);
    }
    else
        std::cout << "\n";
}
```

- Podział na pierwszy parametr (head) i resztę (tail)
- Quasi rekurencja (bo każda funkcja jest jednak inna)
- `if constexpr` umożliwia jej zatrzymanie bez uciekania się do specjalizacji szablonu

# Przykład

```
void print(const char* format) {
    std::cout << format;
}

template<typename T, typename... Args>
void print(const char* format,
           T value, Args... args)
{
    for ( ; *format != '\0'; format++ ) {
        if ( *format == '%' ) {
            std::cout << value;
            print(format + 1, args...);
            return;
        }
        std::cout << *format;
    }
}

int main() {
    print("%cie% %\n", "Witaj", '!', 20);
}
```

1 argument

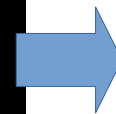
≥ 2 argumenty

Witajcie! 20

# int...

```
template<int NHead, int... NTail>
int sum()
{
    if constexpr (sizeof...(NTail) == 0)
        return NHead;
    else
        return NHead + sum<NTail...>();
}

int main()
{
    std::cout << sum<1>() << "\n";
    std::cout << sum<1, 2, 4, 8, 16>() << "\n";
    std::cout << sum<'1', '2', '4'>() << "\n";
}
```



```
1
31
151
```

# auto...

```
template<auto NHead, auto... NTail>
int sum()
{
    if constexpr (sizeof...(NTail) == 0)
        return NHead;
    else
        return NHead + sum<NTail...>();
}

int main()
{
    std::cout << sum<1, 2, 4, 8, 16>() << "\n";
    std::cout << sum<'1'>() << "\n";
    std::cout << sum<'1', 2, 3llu>() << "\n";
}
```

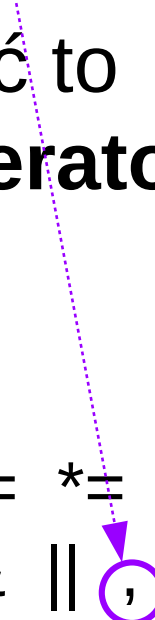


```
31
49
54
```

# *Fold expressions*

- Zwyczajny ***pack expansion*** powoduje rozwinięcie argumentu szablonu do listy z elementami oddzielonymi **przecinkami**
- ***Fold expression*** pozwala uogólnić to rozwinięcie na niemal dowolny **operator dwuargumentowy**

+ - \* / % ^ & | = < > << >> += -= \*= /= %=  
^= &= |= <<= >>= == != <= >= && || , .\* ->\*





# Cztery rodzaje *fold expressions*

## Explanation

The instantiation of a *fold expression* expands the expression  $e$  as follows:

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

(where  $N$  is the number of elements in the pack expansion)

- Wyrażenie  $E$  zawiera nierozwinięty *parameter pack*
- Wyrażenie  $I$  to „inicjalizator”, „wartość początkowa”
- Łączność lewo- lub prawostronna wynika z położenia inicjalizatora (nawet gdy go pominięto)

# Przykład: suma

```
template<typename... Args >
auto sum(Args... args)
{
    return (args + ... + 0);
}

int main()
{
    // std::cout << (1 + (2 + (4 + 0))) << "\n";
    std::cout << sum(1, 2, 4) << "\n";
    // std::cout << (1.1 + (2 + (3.3f + 0))) << "\n";
    std::cout << sum(1.1, 2, 3.3f) << "\n";
}
```

Nawiasy ograniczają zakres rozwinięcia:  
bez return!

- Rozwinięcie wg operatora +
- $E = \text{args}, \quad I = 0$
- ... rozwija  $E$ , zaczynając od strony  $I$

# Suma kwadratów

```
template<typename... Args >
auto sum_sqr(Args... args)
{
    return ((args * args) + ... + 0);
}
    Nawiasy są tu konieczne
int main()
{
    float t = std::sqrt(3.0f);
    double d = std::sqrt(2.0);
    std::cout << sum_sqr(1, 2u, 4ll) << "\n";
    std::cout << sum_sqr(-1, d, t) << "\n";
}
```

21  
6

# Inicjator można pominąć

```
#include <iostream>
#include <string>

template<typename ...Args> auto sum(Args ...args)
{
    return (args + ... + 0);
}

template<typename ...Args> auto sum2(Args ...args)
{
    return (args + ...);
}

int main()
{
    std::cout << sum(1, 2, 3, 4, 5) << "\n";
    std::cout << sum2(1, 2, 3, 4, 5) << "\n";
}
```

# Przykład

```
template<typename ...Args>
void printer(Args&&... args) {
    (std::cout << ... << args) << '\n';
}
```

**Inicjalizator** (*I*)      **Operator** (*op*)      **Wyrażenie** (*E*)

część nierozwijana (nawiasy)

```
printer(1, 2, 3, "abc");
```



$((((I \text{ op } E1) \text{ op } E2) \text{ op } E3) \text{ op } E4) \ll '\n';$



$((((\text{std::cout} \ll 1) \ll 2) \ll 3) \ll \text{"abc"}) \ll '\n';$

# Wariadyczne wyrażenia lambda

- Funkcjonują tak jak wariadyczne szablony

```
#include <iostream>

auto print = [](auto... params)
{
    auto list = {params...};
    for (auto && x: list)
        std::cout << x << "\n";
};

int main()
{
    print(1, 2, 3);
}
```

# Literatura

- <https://arne-mertz.de/2016/11/modern-c-features-variadic-templates/>
- [https://en.cppreference.com/w/cpp/language/parameter\\_pack](https://en.cppreference.com/w/cpp/language/parameter_pack)
- <https://en.cppreference.com/w/cpp/language/fold>