



Uniwersytet
Wrocławski

Szablony w C++

Zbigniew Koza

Wydział Fizyki i Astronomii

Motywacja

- Python:

```
def f(x):  
    print(type(x))  
  
x = 1  
f(x)  
x = 3.14159  
f(x)  
x = "Ala"  
f(x)
```



```
<class 'int'>  
<class 'float'>  
<class 'str'>
```

- **Python** jest językiem z **dynamicznym** systemem typów
- **Łatwo** pisze się w nim **uniwersalne** funkcje działające na obiektach różnych typów

Motywacja

- Python:

```
def f(x):  
    print(type(x))  
  
x = 1  
f(x)  
x = 3.14159  
f(x)  
x = "Ala"  
f(x)
```



```
<class 'int'>  
<class 'float'>  
<class 'str'>
```

- **C/C++** są językami ze **statycznym** systemem typów
- W **C** **trudno** pisze się **uniwersalne** funkcje działające na obiektach różnych typów
- C++ ma rozwiązanie

Motywacja

- Python:

```
def f(x):  
    print(type(x))  
  
x = 1  
f(x)  
x = 3.14159  
f(x)  
x = "Ala"  
f(x)
```



```
<class 'int'>  
<class 'float'>  
<class 'str'>
```

- C++

```
#include <iostream>  
#include <string>  
#include <typeinfo>  
  
template <typename T>  
std::string name(const T & obj)  
{  
    return typeid(obj).name();  
}  
  
int main()  
{  
    auto x = 1;  
    std::cout << name(x) << "\n";  
    x = 3.1345;  
    std::cout << name(x) << "\n";  
    // x = "Ala";  
    std::cout << name("Ala") << "\n";  
}
```



```
i  
i  
A4_c
```

Motywacja

- Python:

```
def f(x):  
    print(type(x))  
  
x = 1  
f(x)  
x = 3.14159  
f(x)  
x = "Ala"  
f(x)
```



```
<class 'int'>  
<class 'float'>  
<class 'str'>
```

- C++

```
#include <iostream>  
#include <string>  
#include <typeinfo>  
  
template <typename T>  
std::string name(const T & obj)  
{  
    return typeid(obj).name();  
}  
  
int main()  
{  
    auto x = 1;  
    std::cout << name(x) << "\n";  
    x = 3.1345;  
    std::cout << name(x) << "\n";  
    // x = "Ala";  
    std::cout << name("Ala") << "\n";  
}
```

```
i  
i  
A4_c
```



```
> ./a.out | c++filt -t  
int  
int  
char [4]
```


Motywacja

- Tak, w C++ jest więcej pisania
- W Pythonie wystarczy napisać jedną funkcję, ale jak ją zoptymalizować?
- W C/C++ dla każdego zestawu typów argumentów należy dostarczyć inną funkcję!
- C: inne nazwy funkcji (np. sin, sinf,...)
- C++: funkcje i klasy generowane są z szablonów (templates)

Podstawy

Szablon

- Definicja:
rozpoczyna się od słowa **template**,
po którym idą parametry w bra-ketach <>
- Zdefiniowanie szablonu nie powoduje
wygenerowania jakiegokolwiek kodu
- Kod jest generowany w momencie użycia,
raz dla każdego nowego zestawu
parametrów szablonu

Szablon

- Definicja:
rozpoczyna się od słowa **template**,
po którym idą parametry w bra-ketach <>
- Zdefiniowanie szablonu nie powoduje
wygenerowania jakiegokolwiek kodu
- Kod jest generowany w momencie użycia,
raz dla każdego nowego zestawu
parametrów szablonu

```
#include <vector> // nie generuje kodu

int main()
{
    std::vector<int> v {1, 2, 3};    // generuje kod
    std::vector<float> w {1, 2, 3}; // generuje kod
    std::vector<int> y {1, 2, 3};    // nie generuje kodu
}
```

```
#include <vector>      // nie generuje kodu
#include <algorithm>    // nie generuje kodu

int main()
{
    std::vector<int> v {1, 3, 2};    // generuje kod
    std::vector<int> w {1, 3};       // nie generuje kodu
    std::vector<float> y {1, 3, 2}; // generuje kod
    std::sort(begin(v), end(v));     // generuje kod
    std::sort(begin(w), end(w));     // nie generuje kodu
    std::sort(begin(y), end(y));     // generuje kod
}
```

Rodzaje szablonów

- Szablony klas
- Szablony funkcji
- Szablony
zmiennych
i stałych

Rodzaje szablonów

- Szablony klas

```
template <typename T>
class vector
{
    size_t size;
    size_t capacity;
    T*      buffer;
public:
    vector(size_t n = 0);
    ...
};
```

Rodzaje szablonów

- Szablony funkcji

```
template <typename T>
inline const T & max(const T & a, const T & b)
{
    return a < b ? b : a;
}
```

Rodzaje szablonów

- Szablony zmiennych i stałych

```
template<class T>
constexpr T pi = T(3.1415926535897932385L);

template<class T>
T circular_area(T r)
{
    return pi<T> * r * r; // pi<T> to szablon zmiennej
}
```

```
auto pi = std::numbers::pi_v<long double>;
```


Rodzaje parametrów szablonów

- Klasy (i typy wbudowane)
- Wartości całkowitoliczbowe

```
#include <array>

std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

Używanie szablonów

- Parametry szablonów mogą podlegać dedukcji lub być podawane jawnie

```
int main()
{
    auto x = std::max(1, 2);           // dedukcja: max<int>
    auto y = std::max(3.0, 4.1);       // dedukcja: max<double>
    auto z = std::max<double>(1, 2.0);
    auto v = std::max<double>(1, 3);
    auto t = std::max<>(1, 3);         // max jest szablonem, dedukcja

    std::vector<int> v0{1, 2, 3};
    std::vector v1{1.0, 3.1};         // dedukcja: vector<double>
    std::array arr = {1, 3, 7};       // dedukcja: array<int, 3>
}
```

Konkretyzacja

- Generowanie kodu klas/funkcji to ich **konkretyzacja**

```
vector<int> v; // możliwa konkretyzacja klasy vector<int>
```

- Szablony sparametryzowany różnymi parametrami reprezentują różne klasy

```
array <int, 5> a;
```

```
array <int, 6> b;
```

```
a = b; // błąd!
```

Kompilacja szablonów

- Treść szablonów musi być znana podczas kompilacji
 - Szablony umieszczaj **w plikach nagłówkowych**
 - Nadawaj im atrybut **inline**
- Kompilacja przeprowadzana jest **w dwóch fazach**:
 - Ogólne sprawdzenie składni (wstępna diagnostyka błędów)
 - Konkretyzacje dla określonych parametrów
- Konkretyzacji podlegają tylko te metody, które tego wymagają
 - Wiele błędów ujawnia się dopiero podczas konkretyzacji
 - Kompilator nie może rozstrzygnąć, czy błąd jest w szablonie, czy w sposobie jego konkretyzacji (użycie niepasującego parametru)

Parametry domyślne

- Bardzo często używane

```
template <typename T, int N = 128>
class array
{
    T tab[N];
    ...
};
```

```
array<int> v; // N = 128
array<int> w; // N = 128
```

Przeciążanie nazw szablonów

- Szablony funkcji (ale nie klas!) mogą być przeciążane


```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

template <typename T>
inline T const& max (const* T const& a, const* T const& b)
{
    return max(*a, *b);
}
```

Specjalizacja częściowa

- Tylko dla szablonów klas


```
template <typename T>
vector
{
    ...
};

// specjalizacja częściowa dla wskaźników
template <typename T>
vector<T*> 
{
    ...
};
```

Specjalizacja pełna

- Dostępna dla szablonów **funkcji** i klas

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// specjalizacja dla const char*
template <> 
inline const char* const& max
    (const char* const& a, const char* const& b)
{
    return strcmp(a,b) == 1 ? b : a;
}
```


- Czy nie lepiej zdefiniować zwykłą funkcję?

Specjalizacja pełna

- Dostępna dla szablonów funkcji i **klas**

```
template <typename T>
struct X
{
    void info() { std::cout << "X" << "\n"; }
};

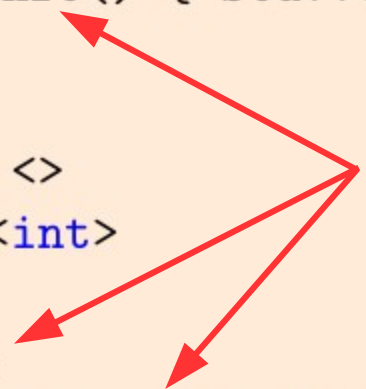
template <>
struct X<int>
{
    int z;
    void info_int() { std::cout << "int jest piękny\n"; }
};
```

Two red arrows originate from the right side of the image and point towards the template parameters of the second struct definition. One arrow points to the empty angle brackets <> in 'template <>', and the other points to the 'int' in 'struct X<int>'. This highlights the specialization of the template.

Specjalizacja pełna

```
template <typename T>
struct X
{
    void info() { std::cout << "X" << "\n"; }
};

template <>
struct X<int>
{
    int z;
    void info_int() { std::cout << "int jest piękny\n"; }
};
```



- Specjalizacje z szablonem ogólnym łączy tylko nazwa

Pułapki składni (zaawansowane)

Zaawansowana składnia

- Definicja wskaźnika czy mnożenie?

```
template <typename T>
class Y {
    ...
};

template <typename T>
class X {
    X x()
    {
        Y<T>::y * z; // definicja czy wyrażenie?
    }
};
```

Definicja wskaźnika czy mnożenie?

```
template <typename T>
class Y {
    ...
};

template <typename T>
class X {
    X x()
    {
        Y<T>::y * z; // definicja czy wyrażenie?
    }
};
```

- Ze względu na możliwość zdefiniowania specjalizacji Y, bez znajomości Y nie można rozstrzygnąć, czy `Y::y` oznacza typ, czy obiekt (składową statyczną).
- Kompilator zakłada, że chodzi o obiekt.
- Użycie składowej Y jako typu wymaga specjalnej składni

Zaawansowana składnia

- **typename** jako deklaratorem typu

```
template <typename T>
class X
{
    typename T::iterator * it; // deklaracja wskaźnika
};
```

- umożliwia odróżnienie deklaracji wskaźnika (ogólnie: typu) od mnożenia (ogólnie: wyrażenia)

```
T::n * b; // mnożenie przez składową statyczną T::n
```

Zaawansowana składnia

```
template <int N>
void print(std::bitset<N> const& bs)
{
    std::cout << bs.template to_string<char> ();
}
```

to_string<char>() jest składową bs

- Konstrukcje **.template**, **::template** i **->template** informują kompilator, że następujący po nich identyfikator reprezentuje **szablon**
- Bez nich kompilator może potraktować <> jako operatory relacyjne
- Używane tylko w szablonach

Zaawansowana składnia

- Dziedziczenie szablonu z szablonu

```
template <typename T>
struct X {
    T size;
};

template <typename T>
class Y : public X<T>
{
    Y() {
        size * x; // skąd wiemy, że klasa bazowa ma składową size?
        this->size * x; // OK
        X<T>::size * x; // OK
    }
};
```

Konieczna nazwa kwalifikowana (**X<T>::** lub **this->**)

Metaprogramowanie

Metaprogramowanie

- Polega na użyciu szablonów oraz reguł gramatyki języka do wymuszenia na kompilatorze wygenerowania kodu źródłowego, skompilowania go i włączenia wyniku do programu
- Może służyć do generowania stałych lub kodu Kodu (klas, funkcji)
- Idea: przeniesienie części pracy z czasu wykonania na kompilator, run time → compile time


Przykład: zliczanie bitów

```
template <unsigned char byte>
struct BITS_SET
{
    enum {
        B0 = (byte & 0x01) ? 1:0,
        B1 = (byte & 0x02) ? 1:0,
        B2 = (byte & 0x04) ? 1:0,
        B3 = (byte & 0x08) ? 1:0,
        B4 = (byte & 0x10) ? 1:0,
        B5 = (byte & 0x20) ? 1:0,
        B6 = (byte & 0x40) ? 1:0,
        B7 = (byte & 0x80) ? 1:0
    };
    enum{RESULT = B0 + B1 + B2 + B3 + B4 + B5 + B6 + B7};
};
...
std::cout << BITS_SET<15>::RESULT;
```

constexpr

- C++ z biegiem lat uzyskał nowe, wygodne narzędzia do zastąpienia metaprogramowania prostszym kodem:
 - **constexpr**
 - **if constexpr**

Przykład: zliczanie bitów



```
#include <iostream>

constexpr int bits_set(unsigned char c)
{
    int result = 0;
    for (int i = 0; i < 8; i++)
    {
        unsigned int mask = 1u << i;
        result += (c & mask) != 0;
    }
    return result;
}

int main()
{
    int x = bits_set(9);
    std::cout << x << "\n";
}
```

Statyczne rozwijanie wyrażeń

- Funkcja → jedna instrukcja asemblera

objdump -C --source a.out

```
int main()
{
```

```
1169: 55
```

```
push %rbp
```

```
116a: 48 89 e5
```

```
mov %rsp,%rbp
```

```
116d: 48 83 ec 10
```

```
sub $0x10,%rsp
```

```
int x = bits_set(9);
```

```
1171: c7 45 fc 02 00 00 00
```

```
movl $0x2, -0x4(%rbp)
```

```
std::cout << x << "\n";
```

```
1178: 8b 45 fc
```

```
mov -0x4(%rbp),%eax
```

```
117b: 89 c6
```

```
mov %eax,%esi
```

```
117d: 48 8d 05 fc 2e 00 00
```

```
lea 0x2efc(%rip),%rax
```

```
# 4080 <std::cout>
```

```
1184: 48 89 c7
```

```
mov %rax,%rdi
```

```
1187: e8 d4 fe ff ff
```

```
call 1060 <std::ostream::operator<<(int)@plt>
```

```
118c: 48 89 c2
```

```
mov %rax,%rdx
```

```
118f: 48 8d 05 6e 0e 00 00
```

```
lea 0xe6e(%rip),%rax
```

```
# 2004 <_IO_stdin_
```

```
1196: 48 89 c6
```

```
mov %rax,%rsi
```

```
1199: 48 89 d7
```

```
mov %rdx,%rdi
```

```
119c: e8 9f fe ff ff
```

```
call 1040 <std::basic_ostream<char, std::char_
```

```
traits<char> >&, char const*)@plt>
```

```
}
```

0x2 = 2 = obliczona wartość

tu jest x

Dygresja: goodbolt

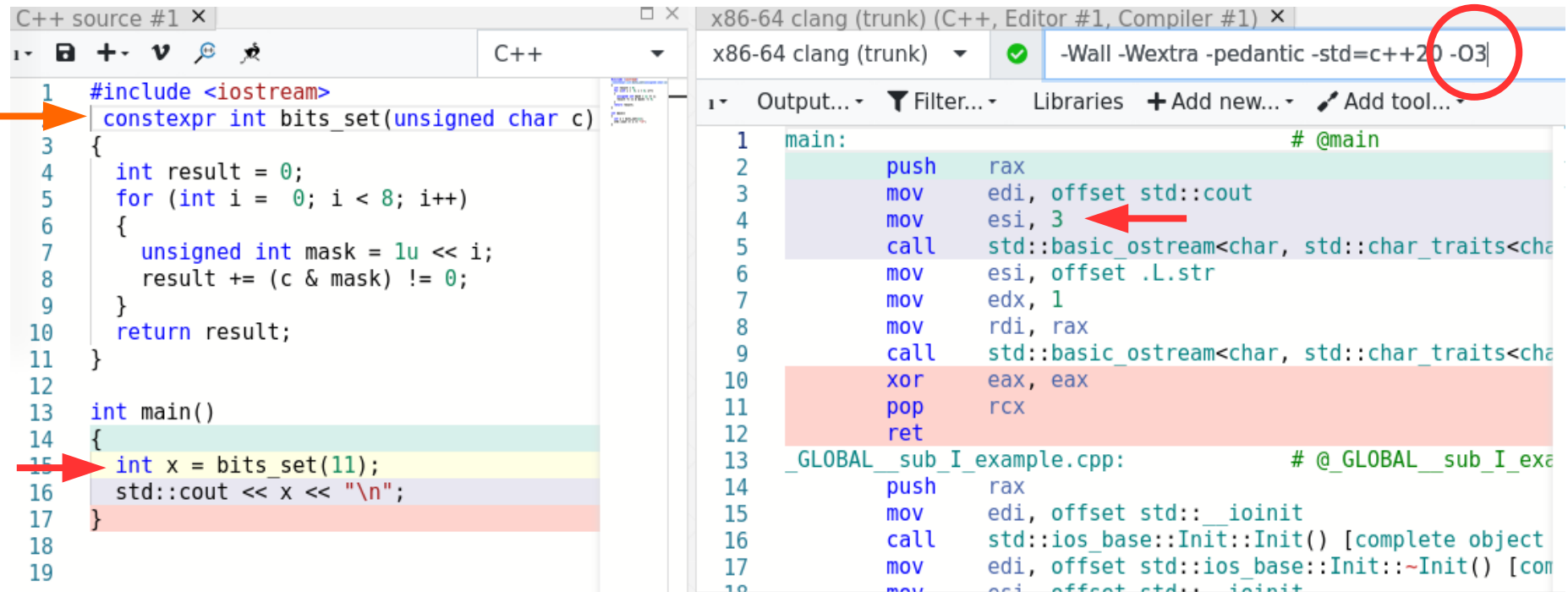
- <https://godbolt.org/z/PoGhsM1aG>

The screenshot displays two panels in Visual Studio Code. The left panel shows a C++ source file named "C++ source #1". It contains a function `bits_set` that takes an unsigned char and returns an integer result based on bit manipulation. Below it is a `main` function that calls `bits_set(11)` and prints the result. Red arrows point to the function call in `main` and the `#include <iostream>` line.

The right panel shows the assembly output generated by x86-64 clang (trunk). The assembly includes standard prologue instructions like `push rbp`, `mov rbp, rsp`, and `sub rsp, 16`. A red arrow points to the instruction `call bits_set(unsigned char)`, which corresponds to the function call in the C++ code. The output also shows stack frame setup for `@main`, including saving registers and setting up local variables.

Dygresja: goodbolt

- <https://godbolt.org/z/PoGhsM1aG>



The screenshot displays the Godbolt compiler explorer interface. On the left, the C++ source code is shown in a window titled 'C++ source #1'. It includes a function `constexpr int bits_set(unsigned char c)` and a `main` function that calls `bits_set(11)` and prints the result. An orange arrow points to the `constexpr` keyword in the function signature. On the right, the assembly output for 'x86-64 clang (trunk)' is shown. The compiler flags `-Wall -Wextra -pedantic -std=c++20 -O3` are visible at the top, with `-O3` circled in red. The assembly for `main` is shown, with a red arrow pointing to the instruction `mov esi, 3`, which corresponds to the argument `11` in the C++ code. The assembly for the `bits_set` function is also visible below.

```
1 #include <iostream>
2 constexpr int bits_set(unsigned char c)
3 {
4     int result = 0;
5     for (int i = 0; i < 8; i++)
6     {
7         unsigned int mask = 1u << i;
8         result += (c & mask) != 0;
9     }
10    return result;
11 }
12
13 int main()
14 {
15     int x = bits_set(11);
16     std::cout << x << "\n";
17 }
```

```
1 main:                                     # @main
2     push    rax
3     mov     edi, offset std::cout
4     mov     esi, 3
5     call    std::basic_ostream<char, std::char_traits<char>>::operator<<@main
6     mov     esi, offset .L.str
7     mov     edx, 1
8     mov     rdi, rax
9     call    std::basic_ostream<char, std::char_traits<char>>::operator<<@main
10    xor     eax, eax
11    pop     rcx
12    ret
13
14 _GLOBAL__sub_I_example.cpp:             # @_GLOBAL__sub_I_example.cpp
15     push    rax
16     mov     edi, offset std::_ioinit
17     call    std::ios_base::Init::Init() [complete object]
18     mov     edi, offset std::ios_base::Init::~~Init() [complete object]
19     call    std::ios_base::Init::~~Init() [complete object]
```

constexpr jest tylko sugestią dla kompilatora

constexpr

- **constexpr** jest tylko wskazówką dla kompilatora
 - Kompilator nie musi jej respektować (zwłaszcza w trybie Debug)
 - Kompilator może włączyć optymalizację „constexpr” nawet bez tego słowa kluczowego (zwłaszcza w trybie Release)
- **constexpr** daje gwarancję, że optymalizacja jest możliwa

SFINAE

SFINAE

- SFINAE = *Substitution failure is not an error*
 - Można przygotować kilka szablonów. Jedne będą kompilować się dla parametrów pożądanых, inne – dla niepożądanych. W ten sposób można odkryć klasę, do jakiej należy parametr szablonu

SFINAE

- *Substitution failure is not an error*

```
// W C++ nie ma tablic o rozmiarze 0:  
// dokładnie jeden z poniższych szablonów zawsze "padnie"  
template<int I>  
bool even(char(*)[I % 2 == 0] = nullptr) { return true; }  
  
template<int I>  
bool even(char(*)[I % 2 == 1] = nullptr) { return false; }  
  
int main()  
{  
    std::cout << even<7>() << even<8>() << "\n"; // 01  
}
```

Substitution *failure is not an error*

```
#include <iostream>

template<typename T>
class IsClass
{
private:
    using One = char;
    using Two = struct { char a[2]; };
    // Konkretyzowana dla C będących klasami
    template<typename C> static One test(int C::*);
    // Konkretyzowana dla C nie będących klasami
    template<typename C> static Two test(...);
public:
    static constexpr bool value =
        sizeof(IsClass<T>::test<T>(0)) == 1;
};

int main()
{
    std::cout << std::boolalpha;
    std::cout << IsClass<int>::value << "\n";
    std::cout << IsClass<IsClass<int>>::value << "\n";
}
```

Wskaźnik
na składową



false
true

enable_if upraszcza SFINAE

- **enable_if** warunkowo usuwa przeciążenie funkcji lub specjalizację szablonu

```
#include <iostream>

template<typename T>
std::enable_if_t<std::is_integral<T>::value> f(T t)
{
    std::cout << "wywołane z argumentem całkowitym\n";
}

template<typename T>
std::enable_if_t<std::is_floating_point<T>::value> f(T t)
{
    std::cout << "wywołane z argumentem zmiennopozycyjnym\n";
}

int main()
{
    f(1);
    f(2.0);
}
```

if constexpr upraszcza SFINAE

- **if constexpr** warunkowo usuwa z **szablonów** gałęzie kodu

```
#include <iostream>

template<typename T>
void f(T t)
{
    if constexpr(std::is_integral<T>::value)
        std::cout << "argument całkowity\n";
    else if constexpr (std::is_floating_point<T>::value)
        std::cout << "argument zmiennopozycyjny\n";
    else
        std::cout << "inny argument\n";
}

int main()
{
    f(1);
    f(2.0);
    f(nullptr);
}
```

Wsparcie biblioteki standardowej

- **#include <type_traits>**
 - Ponad 100 szablonów wspomagających programowanie generyczne
 - `is_integral`, `is_pointer`, `is_arithmetic`,...
 - `is_const`, `is_abstract`, `is_unsigned`,...
 - `has_virtual_destructor`,...
 - `is_same`, `is_base_of`,...
 - `remove_reference`, `add_const`,...
 - `enable_if`,...
 - ...

Szablony wariadyczne

Variadic template

- Przykład:

```
template<typename T, typename ...Args>
T sum(T first, Args... args)
{
    if constexpr(sizeof...(args) == 0)
        return first;
    else
        return first + sum(args...);
}

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

(będzie o tym osobny wykład)

Koncepty (C++20)

Problem: statyczna kontrola typów

- C++ jest językiem z **silną statyczną kontrolą typów**
 - W wyrażeniach (np. $x + y$)
 - W przekazywaniu argumentów do funkcji
- Czy można tą kontrolę objąć **szablony**?
template <typename T> ...
 - \approx żaden szablon nie działa na dowolnym typie T
 - Jak zawęzić **zbiór** dopuszczalnych dla danego szablonu typów T, by błędy użycia szablonów wyłapywać podczas kompilacji?

Koncepty - przykład

```
template <typename T>
concept HasSubscriptOperator = requires(T a)
{
    a[0]; // to wyrażenie musi być poprawne
};

template <HasSubscriptOperator T>
auto front(const T & t)
{
    return t[0];
}
```

- Nowe słowa kluczowe: **concept**, **requires**
- Możliwość definiowania warunków, jakie musi spełniać każdy parametr szablonu
- Łatwiejsza diagnostyka błędów

Koncepty - przykład

```
template <typename T>
concept HasSubscriptOperator = requires(T a)
{
    a[0]; // to wyrażenie musi być poprawne
};

template <HasSubscriptOperator T>
auto front(const T & t)
{
    return t[0];
}
```

```
int main()
{
    std::vector v = {1, 2, 4, 8};
    std::cout << front(v) << "\n";
    // std::cout << front(1) << "\n";
}
```

- Nowe słowa kluczowe: **concept**, **requires**
- Możliwość definiowania warunków, jakie musi spełniać każdy parametr szablonu
- Łatwiejsza diagnostyka błędów

Ułatwienia diagnostyki

- Błąd:

```
int main()
{
    const int x = 7;
    const int y = 1;
    std::swap(x, y);
}
```

- g++ -std=98:

```
In file included from /usr/include/c++/11.1.0/bits/stl_pair.h:59,
                  from /usr/include/c++/11.1.0/bits/stl_algobase.h:64,
                  from /usr/include/c++/11.1.0/bits/char_traits.h:39,
                  from /usr/include/c++/11.1.0/ios:40,
                  from /usr/include/c++/11.1.0/ostream:38,
                  from /usr/include/c++/11.1.0/iostream:39,
                  from 16.cpp:1:
/usr/include/c++/11.1.0/bits/move.h: In instantiation of 'void std::swap(_Tp&, _Tp&) [with _Tp = const int]':
16.cpp:8:12:   required from here
/usr/include/c++/11.1.0/bits/move.h:205:11: error: assignment of read-only reference '__a'
 205 |         __a = _GLIBCXX_MOVE(__b);
      |         ^
/usr/include/c++/11.1.0/bits/move.h:206:11: error: assignment of read-only reference '__b'
 206 |         __b = _GLIBCXX_MOVE(__tmp);
      |         ^
```

Błąd błędnie wykazywany jest w definicji szablonu

Ułatwienia diagnostyki

- Błąd:

```
int main()
{
    const int x = 7;
    const int y = 1;
    std::swap(x, y);
}
```

- clang++ -std=20:

```
16.cpp:8:3: error: no matching function for call to 'swap'
```

```
    std::swap(x, y);
```

```
    ^~~~~~
```

```
/usr/bin/../lib64/gcc/x86_64-pc-linux-gnu/11.1.0/../../../../include/c++/11.1.0/bits/move.h:196:5:
```

```
note: candidate template ignored: requirement '__and_<std::__not_<std::__is_tuple_like<const int>>,
```

```
std::is_move_constructible<const int>, std::is_move_assignable<const int>>::value' was not satisfi
```

```
ed [with _Tp = const int]
```

```
    swap(_Tp& __a, _Tp& __b)
```

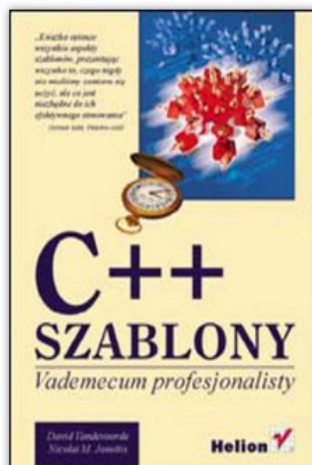
```
    ^
```

- Błąd **prawidłowo** wykazywany w definicji szablonu
- Lista niespełnionych wymagań (**requirements**)

Koncepty

- gcc, clang mają je zaimplementowane od wielu lat
 - Teraz są w standardzie
 - W niezrozumiałych komunikatach diagnostycznych kompilatora szukaj słów *requirement* i/lub *concept*
- We własnym kodzie możesz łatwo wyeliminować błędy związane z nieplanowanym (i nieprzetestowanym) użyciem własnych szablonów

Literatura



C++. Szablony. Vademecum profesjonalisty

Autorzy: **David Vandevorde, Nicolai M. Josuttis**

Niedostępna

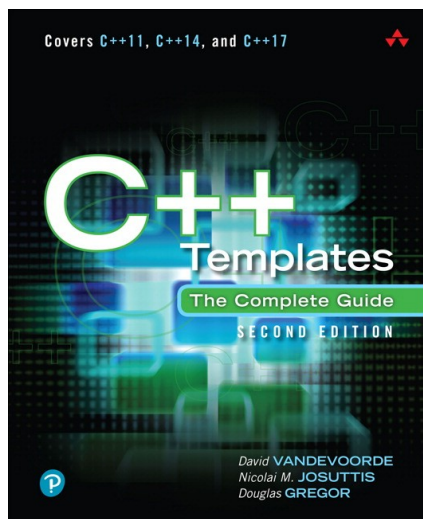
Wydawnictwo: Helion

Ocena: ★★★★★ 6.0/6 | Opinie: 2 ☹

Stron: 480

Druk: oprawa twarda

- 480 stron, C++98



- 822 strony, C++17