Szablony w **C++**

Adrian Wysocki, Kamil Warchoł, Michał Szczepańczyk

20 marca 2018

Szablony

- szablon - sposób na ogólne zapisanie funkcji lub klasy
- dzięki szablonom możliwe programowanie uogólnione
- instrukcja dla kompilatora, jak stworzyć definicje
- automatyzacja procesu generowania roznych odmian funkcji i klas
- nie skracają kodu wynikowego
- szablon klasy zawsze musi znaleźć się w zasięgu globalnym
- przydatne, gdy potrzebne nam są funkcje używające tego samego algorytmu, lub klasy o takiej samej strukturze, ale dla innego typu danych

Tworzenie szablonu klasy cz.1

- słowo kluczowe template - informacja dla kompilatora, że definiujemy szablon
- typename / class
- lepiej używać typename
- przykładowe definiowanie szablonu klasy:

```
template<typename Nazwa_Typu>
class Nazwa_Klasy
{
  Nazwa_Typu m_tab[9999];
}
```

Tworzenie szablonu klasy cz.2

- metody klasy szablonowej, można definiować w klasie

```
template<typename Nazwa_Typu>
class Nazwa_Klasy
{
   ...
   Nazwa_Typu m_element;
   Nazwa_Typu getElement() { return m_element; }
   ...
}
```

Tworzenie szablonu klasy cz.3

- metody klasy szablonowej, można definiować też poza klasą

```
template<typename Nazwa_Typu>
class Nazwa_Klasy
{
   ...
   Nazwa_Typu m_element;
   Nazwa_Typu getElement();
}
template<typename Nazwa_Typu>
Nazwa_Typu Nazwa_Klasy<NazwaTypu>::getElement()
{
   return m_element;
}
```

Użycie szablonu klasy cz.1

- szablonów nie można osobno skompilować
- mając zdefiniowany szablon klasy, nadal nie mamy definicji klasy
- trzeba utworzyć jej konkretyzację:

```
int main()
{
   ...
   Nazwa_Klasy<Typ1> nazwa_obiektu_1;
   Nazwa_Klasy<Typ2> nazwa_obiektu_2;
   ...
}
```

Użycie szablonu klasy cz.2

- kompilator napotykając taką konstrukcję stworzy dwie odrębne definicje klasy i metod

przyklad_1.1

- w przykładzie zakładamy, że możliwe jest przypisywanie jednego elementu do drugiego
- w powyższych przykładach Nazwa_Typu to "parametry typowe szablonów"

Użycie szablonu klasy cz.3

- trzeba jawnie podawać dany typ dla szablonów klas
- dla funkcji nie trzeba było tego robić ponieważ kompilator sam mógł sprawdzić typ

```
template<typename T1, typename T2>
void foo(T1 arg1, T2 arg2) {...}

int main()
{
  int x = 1;
  double y = 5.25;
  foo(x, y);
}
```

Ograniczenia Szablonów Klasy

- trudno jest napisać szablon, który będzie działał dla każdego typu
- podawanie typu na którym nie będą działały niektóre metody
- niepoprawne stosowanie szablonów z parametrem wskaźnikowym

```
template<typename T>
struct Wrapper
{
  T m_value;
  bool operator>(T &wrapper)
  {
    return m_value > wrapper.m_value;
  }
}
```

Argumenty pozatypowe szablonu cz.1

- parametrem dla szablonu nie musi być typ
- argument pozatypowy musi być liczbą całkowitą, typem wyliczeniowym, wskaźnikiem lub referencją

```
template<typename T, int SIZE>
Class Array
{
   ...
}
```

przyklad_1.2

Argumenty pozatypowe szablonu cz.2

- nie mozna pobierac adresu argumentow pozatypowych
- nie mozna modyfikowac argumentow pozatypowych

```
template<typename T, int SIZE>
Class Array
{
   ...
   Array()
   {
      SIZE++;        // ZLE
   }
   ...
}
```

Argumenty pozatypowe szablonu cz.3

- jakie są zalety, a jakie wady takiego podejścia?

```
int main()
{
    ...
    std::array<double, 4> arr1;      // stos
    std::array<double, 6> arr2;
    ...
    std::vector<double> vec1(4);   // sterta
    std::vector<double> vec2(6);
    ...
}
```

Argumenty pozatypowe szablonu cz.4

- jako parametr do szablonu można również przekazać... inny szablon

```
template<template<typename T> class Nazwa_1>
class Nazwa_2
{
   ...
   Nazwa_1<int> m_element;
   ...
}
```

- o tym dokładnie później

Duża elastyczność szablonów

- na klasach szablonowych można używać tych samych technik co na zwykłych klasach
- mogą być klasą bazową
- mogą być klasą pochodną
- mogą być zawierać się w innej klasie
- mogą być argumentem dla innych szablonów
- mogą być używane rekurencyjnie:

przykład_1.3

```
int main
{
   std::array<std::stack<Inna_Klasa>, 10> Array;
   ...
   std::stack<std::stack<int>> Stack;
}
```

Więcej parametrów będących typem

- można przesyłać więcej niż jeden typ
- np. w bibliotece STL szablon "pair"

```
int main()
{
  pair<int, char> PAIR1 ;

  PAIR1.first = 100;
  PAIR1.second = 'G' ;

  cout << PAIR1.first << " " ;
  cout << PAIR1.second << endl ;

  // 100 G
  return 0;
}
```

Parametry domyśle szablonu

- podobnie jak dla funkcji, czy metod można określić domyślne parametry szablonu
- często używane w implementacji STL'a

```
template <typename T1, typename T2 = double>
class Nazwa_Klasy {...};

int main()
{
   ...
   Nazwa_Klasy<int> object; // object jest typu
   ...                      // Nazwa_klasy<int, double>
}
```

Specjalizacja szablonu

- specjalizacje szablonu są podobne do specjalizacji funkcji
- podobnie jak dla funkcji możemy mieć:
- niejawne konkretyzacje
- jawne konkretyzacje
- jawne specjalizacje
- specjalizacje częściowe

Konkretyzacja niejawna

- konkretyzacja niejawna występowała w każdym przykładzie to tej pory

```
template <typename T1>
class Nazwa_Klasy {...};

int main()
{
   ...
   Nazwa_Klasy<int> object;
   Nazwa_Klasy<double> object;
   ...
}
```

- kompilator generuje definicję, na podstawie szablonu

Konkretyzacja jawna

- można wymusić na kompilatorze konkretyzację szablonu

```
template <typename T1>
class Nazwa_Klasy {...};

template class Nazwa_Klasy<parametry>;
```

- nie powstaje żaden obiekt
- wygenerowana jest definicja klasy razem z metodami

Specjalizacja jawna

- definicja szablonu dla konkretnego typu
- nie używany jest wtedy szablon ogólny
- niezbędne, gdy dla danego typu danych szablon ma działać inaczej

```
template <typename T1>
class Nazwa_Klasy {...};

template <>
class Nazwa_Klasy<typ> {...};
```

Specjalizacja częściowa cz.1

- udostępnia konkretny typ dla jednego z parametrów będącego typem:
- którą wersję wybierze kompilator?

```
/// ogolny szablon
template <typename T1, typename T2>
class Nazwa_Klasy
{
   ...
};

/// przykladowa specjalizacja czesciowa
template <typename T>
class Nazwa_Klasy<T, double>
{
   ...
};
```

przykład_1.4

Specjalizacja częściowa cz.2

- można zdefiniować również specjalizację częściową dla typu wskaźnikowego

```
/// ogolny szablon
template <typename T1>
class Nazwa_Klasy { ... };

/// specjalizacja czesciowa dla wskaznikow
template <typename T>
class Nazwa_Klasy<typename T1*> { ... };
```

- jeśli nie ma zdefiniowanej wersji dla wskaźników, kompilator wygeneruje klasę dla szablonu ogólnego(często niebezpieczne) przykład_1.5

decltype (C++ 11) cz.1

- od standardu C++11 dostępne jest słowo kluczowe: decltype
- używa się go w następujący sposób:

```
double x;
decltype(x) y;
```

- w powyższym przykładzie decltype zwraca typ zmiennej x, czyli double i tworzy zmienna y, typu double
- decltype może przyjmować również wyrażenia:

```
decltype(x + y) z = x + y;
```

decltype (C++ 11) cz.2

- dzięki decltype, możliwa jest opóźniona deklaracja typu zwracanego, gdy piszemy funkcję szablonową
- czy można to zrobić w ten sposób?

```
template<typename T1, typename T2>
decltype(x + y) add(T1 x, T2 y)
{
  return x + y;
}
```

decltype (C++ 11) cz.3

- nie można, ale da się zrobić to nieco inaczej
- nagłówek funkcji:

```
double add(int x, float y);
```

- można zapisać jako:

```
auto add(int x, float y) -> double;
```

- analogicznie dla szablonu funkcji:

```
template<typename T1, typename T2>
auto add(T1 x, T2 y) -> decltype(x + y)
{
  return x + y;
}
```

Statyczny polimorfizm

- polimorficzne wywołanie funkcji nie musi być odbywać się poprzez użycie funkcji wirualnych
- przy polimorfiźmie statycznym kompilator decyduje na podstawie typu, którą wersję funkcji wybrać
- nie trzeba używać ani wskaźników, ani referencji
- brak wspólnej hierarchii dziedziczenia
  przyklad_1.6a
  przyklad_1.6b

Typy domyślne

- Typ w deklaracji typów szablonowych może mieć typ domyślny
- Dotyczy to **tylko i wyłącznie** szablonów klas (do C++11)

```
template<typename T = int>
class Collection {
    T * data;
public:
    explicit Collection(std::size_t size)
    : data(new T[size]) {}
    ...
};

Collection<double> double_col(20);
Collection<> def_col(30);
```

Typy domyślne - ograniczenia

- Typy domyślne mogą zostać zadeklarowane tylko dla typów znajdujących się po prawej stronie – analogicznie jak wartości domyślne w funkcjach

```
template<typename First, typename Second = double,
         typename Last>          // zle

template<typename First, typename Second = double,
         typename Last = int> // dobrze
```

Parametry "nietypowe" – non-type parameters

- Parametrami szablonów nie muszą być typy, mogą być to również wartości całkowite
- Jednakże ich wartość musi być znana już podczas **kompilacji**

```
template<typename T, std::size_t size>
class Collection {
    T data[size];
    ...
};

Collection<unsigned long, 10> col; // dobrze

int i = 20;
Collection<double, i> bad_col;      // zle
```

Dozwolone typy

- Dozwolonymi typami w poza-typowych parametrach są (opcjonalnie z kwalifikatorami cv):
  - typy całkowite i wyliczeniowe (enum)
  - wskaźnik do obiektu albo funkcji
  - referencja do l-wartości
  - wskaźnik do składnika klasy (w tym funkcji składowej)
  - std::nullptr_t

Przykład 2.1

Aliasy typów

- Od standardu C++11 istnieje alternatywna składnia dla słowa kluczowego typedef

```
typedef std::vector<int, Alloc<int>> vec_int;
vec_int v;  // std::vector<int, Alloc<int>> v;

using vec_dbl = std::vector<double, Alloc<double>>;
vec_dbl vd; // std::vector<double, Alloc<double>> vd;
```

- Ogólna składnia wygląda następująco:

```
using alias_typu = nazwa_typu;
```

Aliasy szablonowe

- Oprócz alternatywnej składni aliasy mogą być szablonowane
- Dzięki temu możemy stworzyć wygodne aliasy zależne od typu

```
template <typename T>
using ptr = T*;

int x;
ptr<x> = &x;
ptr<const char> = "Hello, world!";
```

Przykład 2.2

Zmienne szablonowe (C++14)

- Przed standardem C++14 "zmienne szablonowe" były opakowywane w klasy

```
template <typename T>
struct is_void {
    static constexpr bool value = ...;
};

std::cout << std::boolalpha << is_void<int>::value
          << std::endl;
```

Zmienne szablonowe

- Wykorzystując zmienne szablonowe możemy uzyskać ten sam efekt pisząc mniej kodu
- Jest to tzw. "lukier składniowy" (syntactic sugar) - pod spodem wszystko działa jakby zmienna była statycznym polem klasy szablonowej

```
template <typename T>
constexpr bool is_void_v = ...;

std::cout << std::boolalpha << is_void_v<int>
          << std::endl;
```

Variadic templates

- Variadic templates to funkcje/klasy szablonowe przyjmujące zmienną liczbę argumentów
- Język C i C++ wspierał funkcje ze zmienną liczbą argumentów, chociażby printf()
- Funkcje takie były jednak niebezpieczne – brak sprawdzania poprawności typów
- Od C++11 możliwe jest tworzenie funkcji szablonowych przyjmujących zmienną liczbę argumentów
- Dodatkowo typ każdego argumentu może być inny

Variadic templates – składania

- Wielokropek określa paczkę parametrów
- Paczka parametrów może zawierać dowolną liczbę parametrów, włącznie z zerem

```
template <typename... Args>
void do_sth(Args... args) {
    ...
}

template <typename First, typename... Rest>
struct Tuple {
    ...
};

Tuple<int, double, std::string> three_elems;
Tuple<long> one_elem;
```

Przykład 2.3

Variadic templates

- Język nie posiada składni do rozpatrywania pojedynczych elementów z paczki parametrów
- Jednakże istnieje możliwość znalezienia liczby przesłanych parametrów za pomocą operatora sizeof...

```
template <typename... Args>
void variadic(Args... args) {
    std::cout << sizeof...(args) << std::endl;
}

variadic(1, 2, 3, 'a'); // wypisze 4
```

Variadic templates – rozpakowywanie

- Wielokropek przed nazwą paczki rozpakowywuje parametry do listy oddzielonej przecinkiem
- Można to wykorzystać w kilku kontekstach

```
template <typename... Bases>
class Derived : public Bases... {
    Derived(const Bases&... bases) : Bases(bases)... {
    ...
    }
};

Derived<B1, B2, B3> derived1 { B1{}, B2{}, B3{} };
Derived<B4, B5>     derived2 { B4{}, B5{} };
```

Variadic templates – fold expressions

- Istnieje możliwość utworzenia wyrażenia zawierającego wszystkie parametry
- Wymaganiem jednak jest posiadanie kompilatora wspierającago standard C++17

```
template <typename... Args>
auto expr(Args... args) {
    return ((2 * args) + ...);
```

- W powyższym przykładzie wyrażenie zostanie rozwinięte do postaci (2 * arg1) + (2 * arg2) + (2 * arg3) + ...

Przykład 2.4

Fold expressions – składnia

- Składnia wygląda następująco:
  - (pack op ...)
  - (... op pack)
  - (pack op ... op init)
  - (init op ... op pack)
- Dozwolonymi operatorami są:
  - dwuargumentowe operatory arytmetyczne
  - dwuargumentowe operatory bitowe
  - dwuargumentowe operatory relacyjne
  - operator przecinek, .*, ->*

Variadic templates – rekurencja

- Aby wykonać bardziej skomplikowane operacje na każdym argumencie z osobna musimy wykorzystać rekurencję
- Oprócz tego funkcja powinna przyjmować co najmniej jeden argument niepochodzący z paczki argumentów
- Dodatkowo należy zapewnić warunek zatrzymania wywołań rekurencyjnych
- W tym celu stosuje się zwykłą, nieszablonową funkcję, która nie przyjmuje parametrów

Przykład 2.5

Klasy parametryzowane wytycznymi

- Klasy parametryzowane wytycznymi (policy classes) to technika konstruowania klas przy pomocy szablonów, dzięki której takie klasy mogą mieć dużo bardziej elastyczne wykorzystanie
- Wytyczne to nic innego jak klasy, które będą określać zachowanie klasy projektowanej
- Z tej techniki korzystają głównie twórcy bibliotek
- W bibliotece standardowej kolekcje są parametryzowane wytycznymi – alokatory

```
template <typename T, typename ErrorHandling = NoChecking<T>>
class SmartPtr : public ErrorHandling {
    ...
};

SmartPtr<int> ptr1;
SmartPtr<int, StrictChecking<int>> ptr2;
```

Przykład 2.6

Szablonowe parametry szablonów

- W poprzednim przykładzie musieliśmy jawnie przekazać parametr do szablonu klasy ErrorHandling:

```
SmartPtr<int, StrictChecking<int>> ptr;
```

- Nic nie stoi na przeszkodzie aby użytkownik podał drugi parametr w zły sposób:

```
SmartPtr<int, SomeOtherClass> ptr1;
SmartPtr<int, StrictChecking<double>> ptr2;
```

- Z pomocą przychodzą nam szablonowe parametry szablonów

Szablonowe parametry szablonów

- Składnia szablonowych parametrów szablonów wygląda następująco (w zastosowaniu do naszego przykładu):

```
template <typename T,
          template <typename U> class ErrorHandling = NoChecking>
class SmartPtr : public ErrorHandling<T> {
    ...
}

SmartPtr<int, SomeOtherClass> ptr1;          // blad kompilacji
SmartPtr<int, StrictChecking<double>> ptr2; // blad kompilacji
SmartPtr<int, StrictChecking<int>> ptr3;    // rowniez blad kompilacji
SmartPtr<int, StrictChecking> ptr4;          // OK
```

- Dzięki takiemu podejściu oddelegowujemy odpowiedzialność za zgodność typów klasy SmartPtr z klasą wytyczną ErrorHandling do definicji klasy SmartPtr

Szablonowe parametry szablonów – pułapki

- Należy jednak wziąc pod uwagę ważny aspekt – ilość parametrów w szablonowym parametrze musi się zgadzać z ilością parametrów przyjmowanych przez dany szablon:

```
template <typename T,
          template <typename> typename Collection> // C++17
class CollectionWrapper {
    Collection<T> col;
    ...
};

CollectionWrapper<int, std::vector> cw; // blad kompilacji,
                                        // std::vector oczekuje dwoch parametrow
```

- Aby to poprawić należy dodać kolejny parametr:

```
template <typename T,
          template <typename, typename> typename Collection>
class CollectionWrapper {
    Collection<T, std::allocator<T>> col;
    ...
};
```

Typy stowarzyszone - cz. 1

- W klasach i funkcjach szablonów możemy za pomocą typedef definiować również typy, nazywane stowarzyszonymi z daną klasą.
- Dzięki temu możemy odwoływać się do nich z innego miejsca

```
template<typename T, int size>
class Stack {
    typedef T value_type;
    value_type x[size];
    ...
};
```

Typy stowarzyszone - cz. 2

```
template<typename Container>
typename Container::value_type sum(Container s) {
    typename Container::value_type total = 0;

    for (int i = 0; i < s.size(); ++i)
        total += s[i];

    return total;
}
```

- Słowo kluczowe **typename** jest tutaj wymagane, bez niego kompilator założyłby że Container::value_type jest zmienną statyczną lub enumem.
- Bez typów stowarzyszonych musielibyśmy przekazać typ elementów kontenera w osobnym argumencie. Dlatego ten mechanizm jest bardzo czesto używany w uogólnionym kodzie.

Przykład 3.1

Type Traits - cechy typów

- są to szablonowe metafunkcje, które zwracają informację o typie w czasie kompilacji
- są dostępne w bibliotece Boost C++, Loki oraz od standardu C++11 znajdują się również w pliku nagłówkowym <**type_traits**> w bibliotece standardowej
- dostępne cechy można podzielić na:
  - sprawdzające kategorie typów (is_pointer, is_enum)
  - sprawdzające własności typów (is_fundamental, is_polymorphic)
  - sprawdzające relacje między różnymi typami (is_same, is_convertible)
  - modyfikujące typ (remove_const, remove_volatile, make_unsigned)
  - inne (enable_if, conditional)
- można również tworzyć własne cechy klas

Jak tworzone są type traits?

- Najczęściej:
  - są zaimplementowane przez kompilator
  - wykorzystują priorytetowość częściowych specjalizacji szablonów
  - wykorzystują zjawisko SFINAE (Substitution Failure Is Not An Error)

Klasy cech - budowa

- Przykładowa klasa type traits:

```
template <typename T>
struct TypeTraits {
    typedef T type;
    const static bool isConst = false;
    enum {
        isPointer = false,
        isRef = false,
    };
};
```

std::integral_constant - podstawowy type traits class

- warto go zapamiętać, ponieważ jest on często wykorzystywany do implementacji wielu innych type traitsów

```
template <typename T, T val>
struct integral_constant {
    static const T value = val;
    typedef T value_type;
    typedef integral_constant<T, val>  type;
    constexpr operator value_type() const noexcept {
        return value; }
};

/// Typ uzywany jako boolean (true) w czasie kompilacji
typedef integral_constant<bool, true>  true_type;
/// Typ uzywany jako boolean (false) w czasie kompilacji
typedef integral_constant<bool, false> false_type;
```

Implementacja is_reference - priorytetowość częściowych specjalizacji szablonów

```
#include <iostream>

template<typename T>
struct my_trait {
    static const bool isReference = false;
//    enum {isReference = false};
};

template<typename T>
struct my_trait<T &> {
    static const bool isReference = true;
//    enum {isReference = true};
};

int main() {
    std::cout << std::boolalpha;
    std::cout << my_trait<int &>::isReference << std::endl; // true
    std::cout << my_trait<int >::isReference << std::endl; // false
}
```

Przykład 3.2

Zastosowanie type traits 1 - warunkowa kompilacja

```
#include <type_traits>

template<typename T,
         typename= typename std::enable_if<std::is_enum<T>::value, void>::type>
void foo(T t) {}

enum Enum1 {A, B};
enum class Enum2 {C, D};

int main() {
    foo<>(A);
    foo(Enum2::C);
//    foo(1); // blad kompilacji - "no matching function for call to 'foo(int)'"
}
```

Przykład 3.3

SFINAE - Substitution Failure Is Not An Error

```
struct Test {
    typedef int foo;
};

template <typename T>
void f(typename T::foo) {} // Definicja #1

template <typename T>
void f(T) {}               // Definicja #2

int main() {
    f<Test>(10); // Wywola #1.
    f<int>(10);  // Wywola #2.
}
```

- kompilator wykonuje int::foo, co powoduje błąd kompilacji, jednak kompilacja nie jest przerywana dzięki SFINAE

Przykład 3.4

Implementacja enable_if - zastosowanie SFINAE

- enable_if może być używane na 3 sposoby w szablonach funkcji:
  - jako typ zwracany zkonkretyzowanej funkcji
  - jako dodatkowy parametr zkonkretyzowanej funkcji
  - jako dodatkowy parametr szablonu

```
template <bool Condition, typename T = void>
struct enable_if {
    // Brak 'type', wiec proba uzycia spowoduje fail substitution
};

// czesciowa specjalizacja, kiedy Condition == true
template <typename T>
struct enable_if<true, T> {
    using type = T;
};
```

Przykład 3.5

Zastosowanie type traits 2 - optymalizacja algorytmu

```
template<typename I1, typename I2, bool b>
I2 copy_imp(I1 first, I1 last, I2 out, const integral_constant<bool, b>&) {
    while(first != last) {
        *out = *first;
        ++out;
        ++first;
    }
    return out;
}

template<typename T>
T* copy_imp(const T* first, const T* last, T* out, const true_type&) {
    memmove(out, first, (last-first)*sizeof(T));
    return out+(last-first);
}

template<typename I1, typename I2>
I2 copy(I1 first, I1 last, I2 out) {
    //
    // We can copy with memcpy if T has a trivial assignment operator,
    // and if the iterator arguments are actually pointers (this last
    // requirement we detect with overload resolution):
    //
    typedef typename std::iterator_traits<I1>::value_type value_type;
    // If a type has a trivial assignment-operator then the operator has the same effect
          as copying the bits of one object to the other: calls to the operator can be
          safely replaced with a call to memcpy.
    return copy_imp(first, last, out, has_trivial_assign<value_type>());
}
```

Przykład 3.6