



Wydział Elektroniki
i Techniki Informatycznych

POLITECHNIKA WARSZAWSKA

Programowanie obiektowe Szablony

Krzysztof Gracki
kgr@ii.pw.edu.pl

pok. 312

**Politechnika
Warszawska**



Tematy

- Tablice z kontrolą indeksów
- Przesłanki dla mechanizmu szablonów
- Szablon klasy **Vect**
- Szablon klasy **Matrix**
- Schemat ogólny szablonów
- Parametry szablonu
- Trzy poziomy abstrakcji
- Szablon stosu
- Szablony klas - wymagania, uwagi
- Konkretyzacje szablonu
- Składowe statyczne w szablonach
- Szablony i zaprzyjaźnienia
- Specjalizacja częściowa szablonu
- Specjalizacja pełna (explicit)
- Specjalizacja funkcji składowych



Politechnika
Warszawska

2



Tablice z kontrolą indeksów (1)

3

Rodzaje tablic w C++

- Tablice **statyczne**, inicjowane domyślnie lub jawnie; rozmiar statyczny, znany w czasie kompilacji
- Tablice **automatyczne**, alokowane na stosie (w ramach bloku); reguły inicjowania – analogiczne; rozmiar statyczny, znany w czasie kompilacji
- Tablice alokowane **dynamicznie** (operator **new []**): inicjacja domyślna, jeśli jest dostępny konstruktor domyślny; brak inicjacji dla typów wbudowanych; rozmiar wyliczany w czasie wykonania
- Odwołania do elementów: $\text{Tab}[\text{ind}] \Leftrightarrow * (\text{Tab} + \text{ind})$

W żadnym z tych przypadków nie ma kontroli zakresu indeksowania

Tablice z kontrolą indeksów (2)

4

```
int main()
{ int T[10];           // Bez inicjalizacji
  static int T1[5];    // Tablica zerowana niejawnie
  int T2[] = { 2, 3, 5, 7 }; // Inicjalizacja jawnie
  Fraction f, f1(1), f2(2,3), f3(f1);
  Fraction Ftab[] = { f, f1, f1, f2, f2 };
  Fraction F[10];      // Inicjalizacja przy pomocy Fraction()
  Fraction *fp=new Fraction[10]; // j.w.
  int *ip = new int[10];
  P(T[5]); // ??, np. -858993460 == 0xCCCCCCCC
  P(T1[2]); // 0
  P(T2[2]); // 5
  P(Ftab[3]); // 2/3
  P(F[5]); // 0
  P(fp[10]); // ??, np. -33686019/-572662307 (INDEKS!)
  P(ip[5]); // ??, np. -858993460 == 0xCCCCCCCC
```

Przykład 1

Tablice z kontrolą indeksów (3)

```
#include<iostream>
using namespace std;
#define P(x) cout << #x" = " <<(x)<< '\n'

struct Punkt { int x, y; };
struct NumName{ int n; const char *nam;};

int Tab[] = {7, 44, 23};
NumName Nid[] = {1,"jeden", 2,"dwa"};
Punkt Mas[][2]= {1,2, 3,4, 5,6, 7,8};
Punkt Mat[][2]=
{ {1,2},{3,4}, { 5,6},{7,8}};
Punkt Mar[][2]= {{{1,2},{3,4}},{5,6},{7,8}}};
ostream& operator<<(ostream& os, const Punkt& p)
{ os<<(' '<<p.x<<',' '<<p.y<<');
  return os;
}
ostream& operator<<(ostream& os, const NumName&
nn)
{ return os<<nn.n<<": " <<nn.nam; }
```

```
int main()
{
  P(sizeof Mas); // 32
  P(sizeof Mat); // 64
  P(sizeof Mar); // 32
  P(Mas[1][1]); // (7,8)
  P(Mat[1][1]); // (0,0)
  P(Mar[1][1]); // (7,8)

  int* Dyn1 = new int [3];
  P(Dyn1[2]); // 1431326031 ??
  int* Dyn1a = new int [3]();
  P(Dyn1a[2]); // 0

  // C++11
  int* Dyn2=new int [3]
  P(Dyn2[2]); // 3
  P(Nid[0]); // 1: jeden
  Tab[3] = 2; // Poza zakresem
  P(Nid[0]); // 2: jeden
}
```

Tablice z kontrolą indeksów (4)

Kontrolę indeksów można wymusić przeciążając operator indeksowania '[]'

```
class IntVect // Klasa dla tablic int
{ int size; // Rozmiar tablicy
  int *s; // Wskazanie na początek tablicy
public:
  IntVect(int n) // KONSTRUKTOR
  { s = new int[n]; size = n; }
  ~IntVect() { delete [] s; } // Destraktor

  int & operator[](int i)
  { assert(i>=0 && i<size);
    // if(i<0||i>=size){ /* Komunikat; */ exit(1); }
    return s[i];
  }
};
```

Przesłanki dla mechanizmu szablonów

7

- Semantyka klasy takiej jak **IntVect** nie zależy od typu elementu:
 - jest dynamicznym kontenerem obiektów
 - ma strukturę liniową z dostępem bezpośrednim
 - podlega stałemu scenariuszowi alokacji i dealokacji
 - potrzebuje detekcji przekroczeń adresu
 - potrzebuje rozróżnienia dostępu modyfikującego i dostępu niemodyfikującego
- Zamiast definiować oddzielne repliki **DoubleVect**, **FractionVect**, ... definiuje się wspólny **szablon klasy (template)** parametryzowany **typem elementów**
- Konkretyzacja szablonu: przez ustalenie typu elementu w deklaracji / definicji obiektu.



Szablon klasy Vect

8

```
template <class Typ> // Szablon klasy
class Vect {
    int n;
    Typ *s;
public:
    Vect(int nn): n(nn), s(new Typ[n]) {}
    ~Vect() { delete [] s; }

    Typ & operator[](int i);
    const Typ & operator[](int i) const;
    int size() {return n; }
    // ...
};
```



```

template<class Typ> // Szablon funkcji skład.
Typ & Vect<Typ>::operator[](int i)
{ assert(i>=0 && i<n);
  // if(i<0 || i>=n) throw ...
  return s[i];
}
template<class Typ>
const Typ & Vect<Typ>::operator[](int i) const
{ assert(i>=0 && i<n);
  // if(i<0 || i>=n) throw ...
  return s[i];
}
void test() {
    Vect<Fraction> F(10);
    Vect<double> D(2*m + 1);
    Vect<int> T(3) = { 2,3,7};
}

```

(cd1) 9



Szablon klasy Vect (cd2)

10

Czy można utworzyć **macierz** pisząc

```
Vect<Vect<int> > M(10); // Wektor wektorów ?
```

Według przyjętej definicji szablonu klasy Vect<> deklaracja jest błędna: **brak konstruktora domyślnego**.


- Deklaracja aktywuje konstruktor, który zleca alokację dynamiczną:
s = new Vect<int> [10];
- Elementy tej tablicy ("bufora") mają typ definiowany przy pomocy klasy, wymagają zatem inicjalizacji przy pomocy **konstruktora domyślnego**
- W klasie **Vect<>** takiego konstruktora nie ma

Konstruktor domyślny można wprowadzić do szablonu przyjmując pewną wartość domniemaną n.

Szablon klasy **Vect** (cd3)

11

```
template <class Typ>
class Vect
{ // enum { STDSIZE = 5 }; // Np. 5
  static constexpr unsigned STDSIZE = 5;
  int n;
  Typ *s;
public:
  Vect(int nn = STDSIZE)
    : n(nn), s(new Typ[n]) {}
  ~Vect() { delete [] s; }
  Typ & operator[](int i);
  const Typ & operator[](int i) const;
  int size() {return n; }
```


 Politechnika
Warszawska



Szablon klasy **Vect** (cd4) //wariant 2

12

```
template <class Typ, int STDSIZE = 5>
class Vect
{ int n;
  Typ *s;
public:
  Vect(int nn = STDSIZE)
    : n(nn), s(new Typ[n]) {}
  ~Vect() { delete [] s; }
  Typ & operator[](int i);
  const Typ & operator[](int i) const;
  int size() {return n; }
};
```

 Politechnika
Warszawska



Szablon klasy **Vect** (cd5)

13

Można pisać:

```
Vect<int> A(10);           // Jak poprzednio
Vect<int> B;               // Jak B(5);
Vect<int, 10> C;           // Jak C(10);
Vect<Vect<int> > D;        // Macierz 5 na 5
Vect<Vect<int, 7> > E;     // Macierz 5 na 7
Vect<Vect<int,7>, 8> F;    // Macierz 8 na 7
Vect<Vect<int>, 8> G(7); // Macierz 7 na 5!!!
```

Bardziej naturalne jest zdefiniowanie bezpośrednio szablonu klasy **Matrix** dla tablic 2 wymiarowych z konstruktorem 2-parametrowymi.

```
template <class Typ>      Szablon klasy Matrix
class Matrix
{ int m,n; // Liczba wierszy i kolumn
  Typ *s;  // Bufor liniowy m*n elementów
public:
  Matrix(int mm = 1, int nn = 1)
    : m(mm),n(nn)
    { s = new Typ[m*n]; }
  ~Matrix() { delete [] s; }

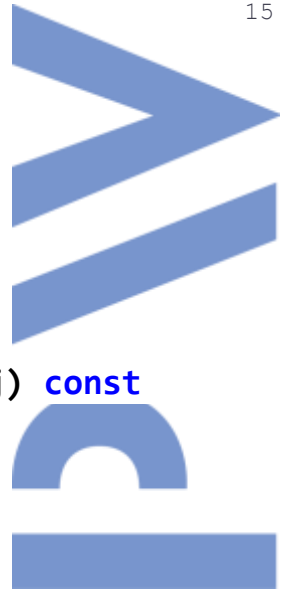
  Typ & operator()(int i, int j);
  const Typ & operator()(int i, int j) const;
  int rows() { return m; }
  int cols() { return n; }
  // ... inne składowe publiczne
};
```

14

Szablon klasy `Matrix` (cd1)

15

```
template <class Typ>
Typ& Matrix<Typ>::operator()(int i, int j)
{
    assert(i>=0 && i<m &&j>=0 &&j<n);
    return s[i*n + j];
}
template <class Typ>
const Typ& Matrix<Typ>::operator()(int i,int j) const
{
    assert(i>=0 && i<m &&j>=0 &&j<n);
    return s[i*n + j];
}
```



Schemat ogólny szablonów

16

```
template <class T> class Node; // Deklaracja
template <class T> class Node // Definicja
{ T *info;
  Node *l, *r; // Node<T> *l, *r;
  // ...
};
template <class Dat, class Res = string>
class X
{ // ...
  Res& convert(const Dat& v); // Deklaracja funkcji w definicji klasy
};
template <class T> // Definicja funkcji
void swap(T&a, T&b){ T t=a; a=b; b=t; }
template <typename T> // Definicja zmiennej szablonowej C++14
T pi = T(3.14159265358979323);
template <> // specjalizacja
const char *pi<const char *> = "pi";
```

```
template < lista_parametrów_szablonu >
deklaracja_lub_definicja_klasy_lub_funkcji
```



Rodzaje parametrów szablonu

17

1. Parametry reprezentujące typy

```
template<class T1, typename T2> // ...
```

2. Parametry reprezentujące wartości

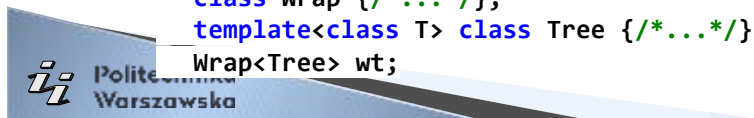
- całkowite, w tym enumeracje
- wskazania i referencje do obiektów lub funkcji
- wskazania na składowe

```
template<typename T, size_t N=1024> class StosN;  
template<class T, int (*gen)()=rand> class Test;  
template<class T, int T::*tip> class Selektor;
```

3. Parametry szablonowe

- specyfikowane jak szablony klas;

```
template< template<typename T> class X>  
class Wrap { /*...*/ };  
template<class T> class Tree { /*...*/ };  
Wrap<Tree> wt;
```



Szablony klas - terminologia

18

Szablon podstawowy
(generyczny)

```
template <class T1, class T2=T1>  
class Tandem { /* ... */ }; // 1
```

Specjalizacja częściowa

```
template <class T>  
class Tandem<T, T*>  
{ /* ... */ }; // 2
```

Specjalizacja pełna

```
template<> class Tandem<int,int*>  
{ /* ... */ }; // 3
```

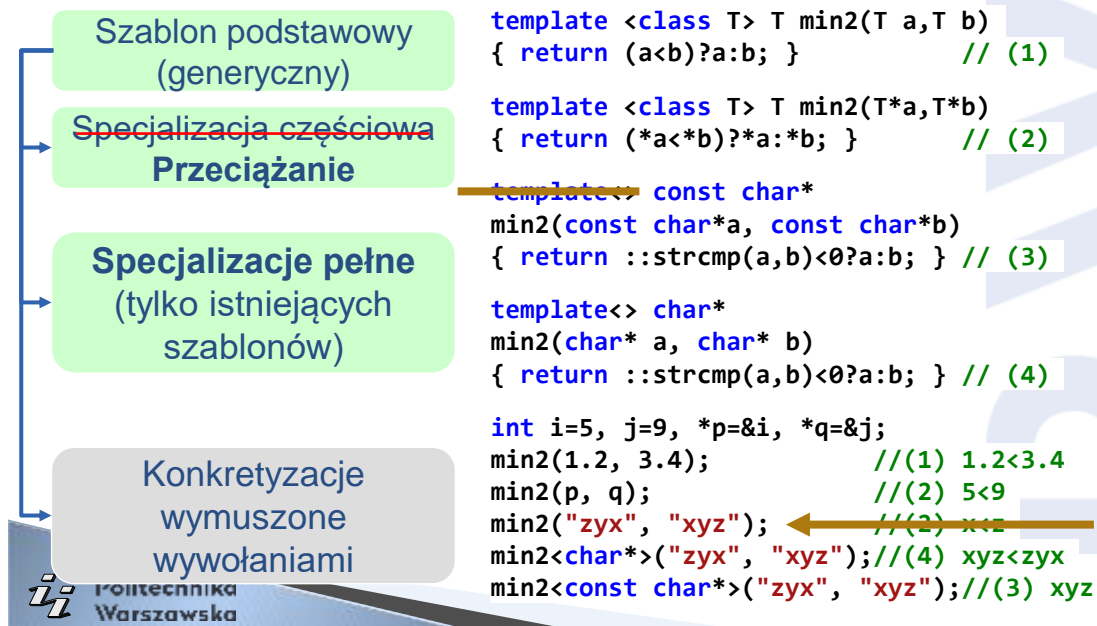
Konkretyzacja
(tworzenie instancji)

```
Tandem<long, bool> tlb; // 1  
Tandem<int> tii; // 1  
Tandem<long, long*> tlp; // 2  
Tandem<int, int*> tip; // 3
```



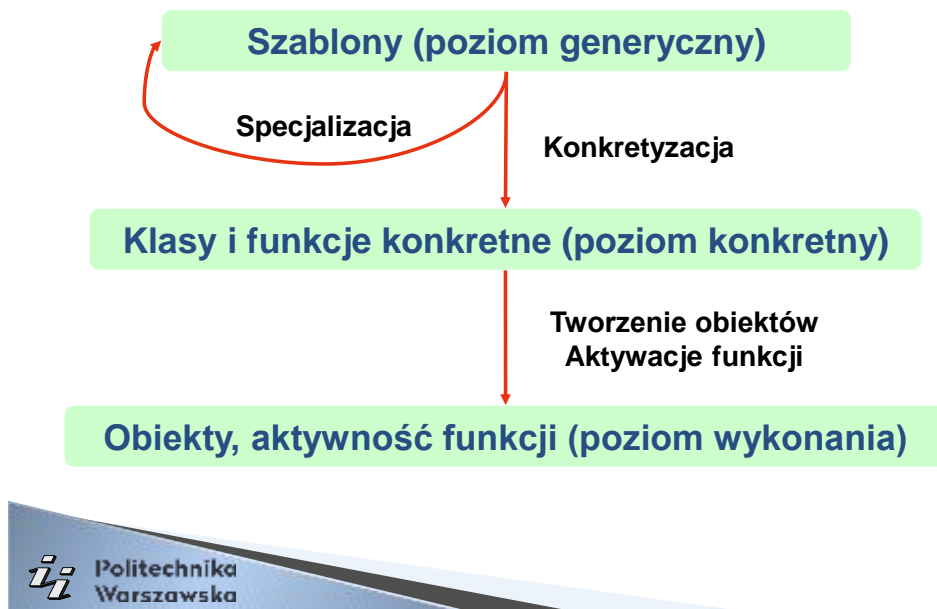
Szablony funkcji - terminologia

19



Trzy poziomy abstrakcji

20



Przykład: szablon stosu

21

```
template <typename T, size_t SIZE = 100>
class Stack
{ int tp;
  T stack[SIZ];
public:
  Stack();
  ~Stack();
  void push(const T& nowy);
  T pop(); // Zdejm i zwróć
  T& top() const; // Podgląd szczytu stosu
  bool empty() const { return tp==0; }
  bool full() const { return tp==SIZE; }
};
```

wartość domyślna

to samo co class

```
template <typename T, size_t SIZE>
Stack<T, SIZE >::Stack() : tp(0) { }

template <typename T, size_t SIZE>
Stack<T, SIZE >::~~Stack() { }

template <typename T, size_t SIZE>
void Stack<T, SIZE >::push(const T& nowy)
{ stack[tp++] = nowy; }

// ...
// Politechnika Warszawska
```

Szablony klas - wymagania, uwagi

22

- Słowo kluczowe **typename** ma także inne zastosowania
- Wewnątrz zasięgu szablonu klasy jawna kwalifikacja nazwami parametrów jest zbędna:

```
template <typename T, size_t SIZE>
class Stack
{ // ...
  Stack(); // zamiast Stack<T, SIZE>();
};
```

Nazwy parametrów w deklaracji i definicji mogą być różne, ale tego samego rodzaju

- Poza klasą podobnie:

```
template <typename Typ, size_t SIZ>
Stack<Typ, SIZ>::~~Stack<Typ, SIZ>(){ }
```

Zbędne

- W konkretyzacji klasy szablonej trzeba użyć typu spełniającego oczekiwania szablonu (czyli z dostępnymi wszystkimi operacjami wywoływanymi na rzecz tego typu)

Konkretyzacje szablonu

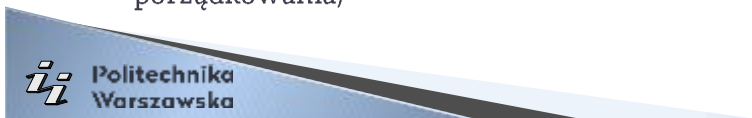
23

- Konkretyzacja (tworzenie instancji klasy wg szablonu) odbywa się wg **strategii minimalistycznej**: kompilator generuje tylko te akcesoria klasy, które są niezbędne. Np.

```
int main()
{ Stack<Fraction> S;
  if(!S.empty()) return 1;
  return 0;
}
```

generuje konstruktor, destruktor i funkcję **empty()**.

- Generacja minimalistyczna zapewnia:
 - mniejsze zapotrzebowanie na pamięć
 - możliwość użycia w konkretyzacji szablonu **typu niespełniającego wszystkich** wymagań semantycznych (np. typ nie definiuje operatora porządku, a w klasie jest metoda porządkowania)



Konkretyzacje szablonu (cd1)

24

- W konkretyzacji parametrowi **całkowitemu** musi odpowiadać wyrażenie stałe:

```
int main()
{ const int N = 1024;
  int n = 1024;
  Stack<Fraction,N> SN; // OK, N jest stałą
  Stack<Fraction,n> Sn; // Błąd
  if(!SN.empty()) return 1;
  return 0;
}
```

- Inne dopuszczalne argumenty konkretyzacji szablonów (dla parametrów różnych od nazwy typu)
 - wskazania na składowe
 - wskazania / referencje na obiekty lub funkcje **z wiązaniem zewnętrznym**
- Wartości typów zmiennopozycyjnych są zakazane



Konkretyzacje szablonu (cd2)

25

```
static char s[] = "ABC";
extern char t[] = "DEF";
template <class T, const char *str> struct A { /*...*/ };
void fun()
{ const char * p = "123";
  A<int, "X"> a1;    // Błąd: literał
  A<int, p> a2;      // Błąd: p ma wiązanie lokalne
  A<int, s> a3;      // Błąd: s ma wiązanie wewnętrzne
  A<int, t> a4;      // OK: wiązanie zewnętrzne
}
```

- Argumentem konkretyzacji szablonu może być konkretyzacja szablonu (p. Vect)
- Jeżeli szablon klasy definiuje **parametry domyślne**, to w konkretyzacji odpowiednie argumenty można pominąć:

```
template <class A, class B, class C = A> class X;
X<Vect<int>, Stack<int>> *xx; // C jest Vect<int>
```

Może być

- Reguły pomijania od prawej - jak dla argumentów funkcji

Konkretyzacje szablonu (cd3)

26

- Jeżeli wszystkie parametry szablonu są predefiniowane, to **konkretyzacja domyślna** wymaga użycia nawiasów <> :

```
template<class T = char> class Buf;
Buf<>* p;    // OK: znaczy Buf<char>*
Buf* q;      // Błąd
Buf<int>* r; // OK
```
- Szablon nie ma żadnych specjalnych przywilejów dostępu do klasy użytej w argumencie konkretyzacji; nazwa klasy w argumencie musi być dostępna w punkcie konkretyzacji:

```
template<class T> class X { /* ... */ };
class Y {
private: struct S { /* ... */ };
  X<S> x; // OK: tu S jest nazwą dostępną
};
X<Y::S> y; // Błąd: tu nazwa S nie jest dostępna
```

Składowe statyczne w szablonach

27

- Składowe statyczne w szablonie klasy mogą być definiowane przy pomocy szablonu; są one konkretyzowane wraz z konkretyzacją klasy:

```
template <class T> class X
{ // ..
    static T sx;
};
template <class T> T X<T>::sx = T();
X<int> xi; // Powstanie int X<int>::sx=0;
X<float> xf; // Powstanie float X<float>::sx=0.0f;
```

Uwaga: Taka kombinacja jest błędna:

```
template<class T> struct A
{ static T t; };
typedef int Fun();
A<Fun> a; // Nie: A<Fun>::t byłoby funkcją statyczną
```

Szablony i zaprzyjaźnienia

28

- Zaprzyjaźnienia nieszablonowe: instancje klasy Vect są zaprzyjaźnione z fun() i klasą X.

```
class X { /*...*/ };
template<class T> class Vect
{ public: //...
    friend void fun();
    friend class X;
};
```

- Zaprzyjaźnienia ze specjalizacją klasy: tylko podana specjalizacja X<char> jest zaprzyjaźniona z Vect

```
template<class T> class X
{ /*...*/ };
template<class T> class Vect
{ public: //...
    friend class X<char>;
};
```

Szablony i zaprzyjaźnienia (cd1)

29

- **Zaprzyjaźnienia szablonowe:** wszystkie specjalizacje **Vect** są zaprzyjaźnione z wszystkimi specjalizacjami klasy **X** i funkcji **f()**

```
template<class W> class X { /*...*/ };
template<class R> int f() { /*...*/ }
template<class T> class Vect
{ public:    //...
    template<class T> friend class X;
    template<class T> friend int f();
};
```



Szablony i zaprzyjaźnienia (cd2)

30

//Przykład: uniwersalne porównywanie wektorów Vect
template<class T> class Vect; // Zapowiedź klasy

template<class T> // Deklaracja szablonu funkcji
bool operator==(const Vect<T>& v1, const Vect<T>& v2);

```
template <class T> class Vect
{ public:    //...
    friend bool operator==<T>(const Vect<T>& v1,
                              const Vect<T>& v2);
};
```

Potrzebny szablon funkcji;
ewentualnie <>

```
template <class T> // Definicja
bool operator== (const Vect<T>& v1, const Vect<T>& v2)
{ if(v1.size() != v2.size()) return false;
  for(int i=0; i<v1.size(); ++i)
    if(v1[i] != v2[i]) return false;
  return true;
}
```



Specjalizacja częściowa szablonu

31

Specjalizacja częściowa szablonu jest to alternatywna definicja dla ustalonej rodziny typów

```
// Szablon pierwotny klasy Vect
template<class T> class Vect { /*...*/};
```

```
// Specjalizacja częściowa Vect
// dla typów wskaźnikowych
```

```
template<class T> class Vect<T*>
{ size_t size; void * p;
public:
```

Specjalizacja częściowa

```
    Vect();
    ~Vect();
    size_t size() const;
    //...inne funkcje składowe
```

Specjalizacja częściowa szablonu (cd1)

32

Dla szablonów wieloparametrowych może być wiele wariantów specjalizacji częściowej, np:

```
// Szablon pierwotny klasy Z
template<class A, class B, int n> class Z{ /*...*/};
```

```
// Specjalizacja częściowa (1)
template<class A, int n> class Z<A, A*, n>{ /*...*/};
```

```
// Specjalizacja częściowa (2)
template<class A> class Z<char*, A*, 10>{ /*...*/};
```

```
// Specjalizacja częściowa (3)
template<class T> class Z<T*, T**, sizeof 5>{ /*...*/};
```

```
// Specjalizacja częściowa (4)
template<class T> class Z<int, Z<T,T,1>, 5>{ /*...*/};
```


Specjalizacja pełna (explicit)

33

Szablon dla pewnego argumentu ma specyficzną definicję odmienną od przypadków specjalizacji częściowych i od definicji pierwotnej (generycznej). Np. `Vect<bool>`:

```
template<> class Vect<bool>
{ size_t n;
  typedef unsigned char uchar;
  uchar *b;
public:
  Vect(size_t nn=1): n(nn), b(new uchar [(n+7)/8]) {}
  Vect<bool>(const Vect<bool> & v);
  Vect<bool>& operator=(const Vect<bool>& v);
  ~Vect<bool>();
  bool& operator[](unsigned int index);
  const bool& operator[](unsigned int index) const;
  size_t size() const;
  // ...
};
```

W bibliotece standardowej jest klasa tego rodzaju:
`std::vector<bool>`
(deprecated)
np. `std::bitset<N>`

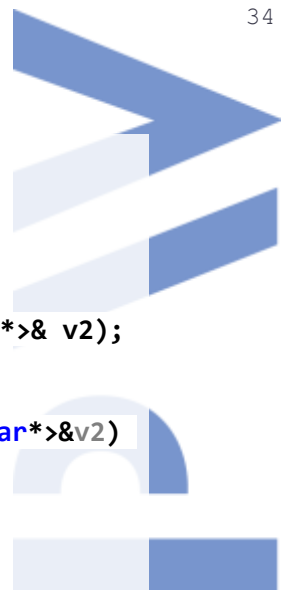


Specjalizacja funkcji składowych

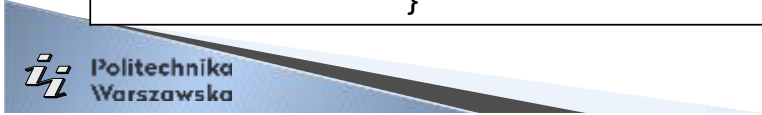
34

Typ argumentu konkretyzacji szablonu może wpływać na sposób realizacji operacji. Np. w klasie `Vect<T>` operacja porównania dla `T=char*` powinna być inna:

```
template<class T> class Vect
{ /* ...*/
  friend bool operator==<T>    // Wersja generyczna
    (const Vect<T>& v1, const Vect<T>& v2);
  friend bool operator==      // Wersja spec.
    (const Vect<const char*>& v1, const Vect<const char*>& v2);
};
template<> bool operator==
  (const Vect<const char*>&v1, const Vect<const char*>&v2)
{ if(v1.size() != v2.size()) // Na pewno różne
  return false;
  for(size_t i = 0; i<v1.size(); ++i)
  if(strcmp(v1[i], v2[i]) != 0) return false;
  return true;
};
```



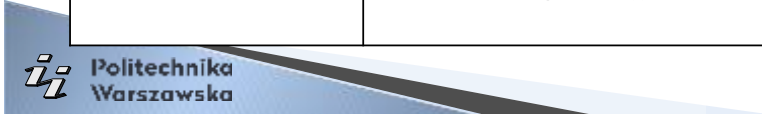
Plik klasa.h	Plik main.cpp
<pre>// Szablon klasy // Funkcje szablonu Klasa<T> #include "klasa.impl" //Plik klasa.impl template<class T> struct Klasa { T get(); const char* getTid(); }; template<class T> T Klasa<T>::get() { return T(); } template<class T> const char* Klasa<T>::getTid() { return typeid(T).name(); }</pre>	<pre>#include "klasa.h" int main() { Klasa<int> ki; ki.get(); ki.getTid(); }</pre>



Konkretyzacje jawne - zastosowanie

Plik klasa.h	Plik klasa.cpp	Plik main.cpp
<pre>// Szablon klasy template<class T> struct Klasa { const char* getTid(); T get(); };</pre>	<pre>#include <typeinfo> #include "klasa.h" // Funkcje szablonu Klasa<T> template<class T> T Klasa<T>::get() { return T(); } template<class T> const char* Klasa<T>::getTid() { return typeid(T).name(); } // jawne konkr. szablonmów template int Klasa<int>::get(); template const char* Klasa<int>::getTid();</pre>	<pre>#include "klasa.h" int main() { Klasa<int> ki; ki.get(); ki.getTid(); }</pre>

Bez umieszczenia konkretyzacji jawnych (wymuszonych) konsolidator zgłasza brak definicji funkcji



Konkretyzacje jawne (wymuszone)

37

Pełna konkretyzacja szablonu może być
wymuszona deklaracją:
template deklaracja_szablonowa

```
template<class T> struct V { T t; void f(); /*...*/ };
template struct V<char>; // Konkretyzacja jawna dla char
namespace N {
    template<class T> void f(T&) { /*...*/ }
}
template void N::f<int>(int&); // Konkretyzacja jawna w N

template <class T> class X {
    T h(T t) { return t; }
};
template int X<int>::h(int); // Konkretyzacja dla int
template<class T> void V<T>::f() // Definicja
{ /*...*/ }
```



auto f(auto a) {return 2 * a; } Szablony i auto C++20

38

```
template<>
auto f<const char*>(const char* a) {
    return "0";
};
auto f(const char* a) {
    // const char* f(const char* a)
    return a;
}

int main()
{
    std::cout << f(10) << std::endl;
    std::cout << f(1.11) << std::endl;
    std::cout << f("A") << std::endl;
}
```

```
template<class type_parameter_0_0>
auto f(type_parameter_0_0 a) {
    return 2 * a;
}

template<>
int f<int>(int a) {
    return 2 * a;
}

template<>
double f<double>(double a) {
    return 2 * a;
}
```

[// https://cppinsights.io/](https://cppinsights.io/)

20
2.22
0

20
2.22
A

