

WYDZIAŁ FIZYKI
UNIWERSYTET WARSZAWSKI

Programowanie zaawansowane FM i NI

Wykład 6

Szablony, static, Lambda

Krzysztof Piasecki

Semestr letni roku akad. 2025-26


krzysztof.piasecki@fuw.edu.pl

- **Argumenty domyślne (default)** funkcji.

  Jeżeli w argumentach wejścia funkcji któremuś z nich przypiszemy wartość, to wywołując funkcję możemy pominąć podanie tego argumentu.

⊙ ***Uwaga***:

jeżeli funkcja w danym argumencie wejścia ma przeciążenie
(jest kilka wariantów argumentu wejścia), to w wywołaniu trzeba dać tam wartość. Inaczej, komputer nie wie, który wariant funkcji ma wywołać. Np. dla:

```
int  Fun (int  a) { return  a; }
char Fun (char b) { return -b; }

int main () {
  cout << Fun() << endl;
}
```

→ próba kompilacji spowoduje błąd.

[Link]

```
#include <iostream>
using namespace std;

int MySum (int a = 1, int b = 2, int c = 3) {
  cout << "[MySum | a, b, c] = "
       << a <<' '<< b <<' '<< c << endl;
  return a + b + c;
}

int main () {
  int X = 4, Y = 5, Z = 6;
  int wynik0 = MySum () , wynik1 = MySum (X),
      wynik2 = MySum (X, Y) ,
      wynik3 = MySum (X, Y, Z);

  cout << "[Main | wariant 0,1,2,3 ] "
       << wynik0 <<' '<< wynik1 <<' '
       << wynik2 <<' '<< wynik3 << endl;
}
```

```
[MySum | a, b, c] = 1 2 3
[MySum | a, b, c] = 4 2 3
[MySum | a, b, c] = 4 5 3
[MySum | a, b, c] = 4 5 6
[Main | wariant 0,1,2,3 ] 6 9 12 15
```

- **Szablon (template)** funkcji.

Poprzedzając nagłówek funkcji składnią:

```
template <typename T>
```

można uogólnić funkcję na **szablon** (**template**)
(nazwa T jest przykładowa i oznacza „jakiś typ”).
W ciele szablonu możemy używać typu (np.) T .

W ten sposób możemy oddzielić algorytm
od typu, na którym ten algorytm pracuje.

W miejscu wywołania szablon potrzebuje poznać
ten typ. Wówczas dokonuje **konkretyzacji**
szablonu (**template instatiation**), czyli dopiero
tu tworzy funkcję pracującą na konkretnym typie.

Sprawdza też, czy na tym typie można pracować
(np. jak funkcja ma dodać, to może dodać 2 liczby
ale nie może dodać 2 wskaźników
lub 2 C-stringów).

Nb: wiele plików nagłówkowych ma szablony.
Dlatego preprocesor je załącza przed kompilacją.
Kompiluje się wtedy i nasz kod, i biblioteka.

[Link]

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename Typ>
5  Typ Max3 (Typ a, Typ b, Typ c)
6  {
7    cout << "[Max3] pracuje na typie: "
8         << typeid( a ).name() << endl;
9    Typ wynik = max ( a , max( b, c) );
10   return wynik ;
11 }
12
13 int main () {
14   int    ia = 1, ib = 2, ic = 3;
15   double da = 3, db = 1, dc = 2;
16
17   auto wynik1 = Max3 (ia, ib, ic);
18   cout << "[main] wynik w typie "
19        << typeid( wynik1 ).name()
20        << " wynosi: " << wynik1 << endl;
21
22   auto wynik2 = Max3 (da, db, dc);
23   cout << "[main] wynik w typie "
24        << typeid( wynik2 ).name()
25        << " wynosi: " << wynik2 << endl;
26 }
```

- **Szablon (template)** funkcji – **priorytety**

  ① W wywołaniu można w <> jawnie podać typ.

  ② W wywołaniu wartości argumentów mają typ

  ③ W deklaracji szablonu można podać typ domyślny.

  ④ W deklaracji w argumentach wejścia można podać wartości domyślne.

⊙ *Jaka będzie **kolejność priorytetów** tych działań podczas konkretyzacji szablonu w funkcję ?*

→ **Typy w wywołaniu** (① i ②) **są najważniejsze**, ale z nich priorytet ma **podanie jawne**, <typ> (①)

→ **W deklaracji domyślny typ** w <...> (③) jest **ważniejszy** od typu argumentów domyślnych (④)

⊙ Podanie tylko tych ostatnich (④) nie wystarcza…

⊙ Nie wolno tu też podać odmiennych typów.

```
s 0
i 0
f 4.25

s 5
d 4.72727
d 5.2
```

[Link]

```
#include <iostream>
#include <typeinfo>
using namespace std;

template<typename Typ>
Typ Fun1 (Typ a = 5.1, Typ b = 1.2)    {
  cout << typeid ( a ).name() << ' ' ;
  return a / b;                         }

template<typename Typ = float>
Typ Fun2 (Typ a = 5 , Typ b = 1 )      {
  cout << typeid ( a ).name() << ' ' ;
  return a / b;                         }

int main () {
  int   ia = 3, ib = 5;
  cout << Fun1<short> (ia, ib) << endl;
  cout << Fun1<>      (ia, ib) << endl;
  cout << Fun1<float> () << endl << endl;
//cout << Fun1        () << endl << endl;

  cout << Fun2<short> ( 5.2, 1.1 ) << endl;
  cout << Fun2        ( 5.2, 1.1 ) << endl;
  cout << Fun2        ( 5.2 )      << endl;
//cout << Fun2<>      ( 5  , 1.1 ) << endl;
}
```

- **Specjalizacja (specialization) szablonu**

  Gdy przewidujemy, że reguła „ogólna”
  nie zadziała dla konkretnego typu
  (a nawet grupy typów),
  to możemy ręcznie dodać wyjątek.
  Nazywamy to **specjalizacją szablonu** .

  W tym przykładzie, funkcja `add` dobrze
  działa na liczbach i `string`ach,
  ale nie potrafi dodać `C-string`ów.
  → dostawiamy wariant dla nich.

- Można utworzyć **szablon na więcej typów**.

  Uwaga: łącząc informacje w deklaracji
  i w miejscu wywołania,
  komputerowi nie może zabraknąć
  pełni wiedzy o typach.
  Inaczej – nie skonkretyzuje szablonu.

```
8
CSTring
ii0
di0.5
dd0.5
```

[Link]

```
#include <iostream>
#include <cstring>
#include <typeinfo>
using namespace std;

template <typename T> T Add (T a, T b)          {
  return a + b ;                                }

template <>       char* Add (char* a, char* b) {
  return strcat ( a , b ) ;                     }

template <typename T1, typename T2>
auto divide (T1 a, T2 b)                        {
  cout << typeid(a).name() << typeid(b).name();
  return a / b;                                 }

int main () {
  cout << Add ( 3, 5 ) << endl;
  char csA[10] = "CST"; char csB[10] = "ring" ;
  cout << Add ( csA , csB )  << endl ;

  cout << divide ( 1  , 2  ) << endl;
  cout << divide ( 1. , 2  ) << endl;
  cout << divide ( 1. , 2. ) << endl;
}
```

Zmienne i funkcje statyczne (`static`)

Specyfikator **static** ma kilka znaczeń.

- użycie do **funkcji i zmiennych globalnych**:

  zakresem tych obiektów będzie aktualny plik kodu.

- użycie do **zmiennych w środku funkcji**:

  Od pierwszego wywołania funkcji zmienna ta przetrwa do końca programu. Przy następnym wywołaniu będzie pamiętać dotychczasową wartość.

  Gdy w linii deklaracji jest przypisanie, to zachodzi ono tylko za pierwszym razem, tj. właśnie przy deklaracji.

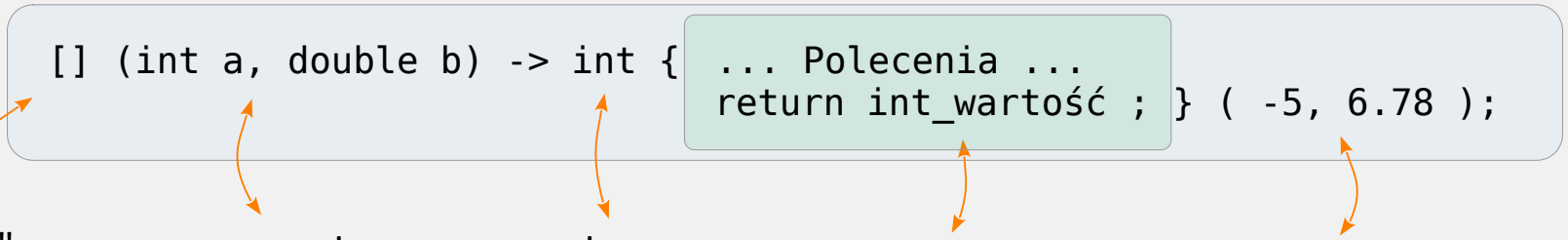
[Link]

```
1  #include <iostream>
2  using namespace std;
3
4  static int statStart = 1;
5
6  static void myStatFun () {
7    cout << "[myStatFun]" << endl;
8  }
9
10 int myCount () {
11   static int N_of_calls = statStart;
12   return ( N_of_calls++ ) ;
13 }
14
15 int main () {
16   cout << "statStart = " << statStart << endl;
17   myStatFun () ;
18
19   for (int i = 1; i <= 3; i++)
20     cout << "[myCount] x" << myCount() << endl;
21 }
```

```
statStart = 1
[myStatFun]
[myCount] x1
[myCount] x2
[myCount] x3
```

- **Wyrażenie lambda** : funkcja anonimowa, którą można zdefiniować w miejscu użycia (np. w innej funkcji)

  Pełna składnia:



- Niektóre elementy można pominąć. Np. tu :

```
cout << [](int a) { return a; } (123) << endl;          →   123
```

kompilator patrząc na `return a` , wydedukuje, że typem zwracanym jest `int` .

A tu:

```
cout << [] () { return 123; } () << endl;               →   123
```

kompilator przyjmie, że lambda nic nie przyjmuje, zaś typ zwracany wydedukuje z typu 123 (int).

- Wyrażenie Lambda można przechować, przypisując je "zmiennej" o typie `auto` :

```
auto Fun1 = [](double x) { return x*exp(-x); } ;

cout << Fun1 (1.) << endl;
```

Tu jeszcze bez `()`, bo tu nie wywołujemy Fun1.

… i następnie je wywołać:

- *Ważne*: Typ obiektu Lambda nie jest żadnym ze znanych typów.
  Każda „sztuka” lambdy dostaje swój ***unikalny*** **typ**. Stąd do przechowania użyliśmy `auto`.

- Od standardu c++20 funkcja w argumentach wejścia może mieć `auto` (oraz `auto&`) .
  Dzięki temu wyrażenie Lambda można przekazać funkcji.

[Link]

Ta funkcja potrafi przyjąć lambdę.

Znak & sprawi, że przyjmie jej oryginał.

```
1  #include <iostream>
2  using namespace std;
3
4  int MyFun (auto& MyLam, int x) {
5    return MyLam ( x ) ;
6  }
7
8  int main () {
9    auto Lam = [] (int x) { return x*x ; } ;
10   cout << MyFun ( Lam, 3 ) << endl;
11 }
```

Tu wołamy funkcję,
podając jej naszą lambdę.

- Lambda potrafi **przechwycić** (**capture**) zmienne z miejsca, w którym została zdefiniowana.

[Link]

Przechwyt c1, read-only

Przechwyt wszystkich, read-only

Nie można zmienić c1, bo jest read-only

Przechwyt c1 przez referencję (można zmienić)

Przechwyt c1 read-only, ale c2 przez referencję.

Przechwyt wszystkich przez referencję.

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5    int x = 3, c1 = 5, c2 = 7;
6
7    cout << [c1] (int x) { return c1 * x; } (2) <<' '<< c2 << endl;
8
9    cout << [=]  (int x) { return c1 * x; } (2) << endl;
10
11 // cout << [c1] (int x) { c1++; return x; } (2) << endl;
12
13   cout << [&c1] (int x) { c1++; return x; } (2) <<' '<< c1 << endl;
14
15   cout << [c1, &c2] (int x) { c2++; return c1 * x; } (2)
16        <<' '<< c2 << endl;
17
18   cout << [&] (int x) { c1++; c2++; return c1 * x; } (2)
19        <<' '<< c1 <<' '<< c2 << endl;
20 }
```