

WYDZIAŁ FIZYKI
UNIWERSYTET WARSZAWSKI

Programowanie zaawansowane FM i NI

Wykład 9

Szablony klas, valarray, iteratory, algorithm

Krzysztof Piasecki

Semestr letni roku akad. 2025-26


krzysztof.piasecki@fuw.edu.pl

- **template class (szablon klasy)**
  – to sposób na uogólnienie typów, na których pracuje klasa.

⊙ Podobnie, jak dla szablonów funkcji, ciało klasy poprzedzamy słowami:

```
template<class MojTyp>
```

(`typename` i `class` są wymienne)

⊙ Obiekt klasy deklarujemy tak:

```
Klasa<Typ> Obiekt ;
```

⊙ Szablon nie jest jeszcze klasą. **Konkretyzacja szablonu do klasy zachodzi w miejscu użycia.** Kompilator sprawdza, czy zakodowane działania na danym typie są możliwe.

Uwaga. Klasa nazywa się wtedy:

`Klasa<Typ>`

Ciekawostka: wiele bibliotek to szablony klas. To dlatego są one kompilowane wraz z naszym kodem.

[Link]

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

template <class T>
class MyData {
  T x;
 public:
  MyData (T _x) : x (_x) { }
  T&   GetX ()     { return x; }
  void SetX (T _x) { x = _x  ; }
};

int main() {
  MyData<int>    i ( 5 )   ; cout << i.GetX() << endl;
  MyData<float>  f (2.4)   ; cout << f.GetX() << endl;
  MyData<char>   c ('t')   ; cout << c.GetX() << endl;

  int q ;
  MyData<int*>  ip ( &q )  ; cout << ip.GetX() << endl;

  MyData< MyData<int> > M ( i ) ;
  cout << M.GetX().GetX() << endl;
}
```

- **template class c. d.**

⊙ Implementacja metod
(w tym konstruktorów i operatorów)
poza ciałem szablonu:

każdą metodę poprzedzamy przez

```
template<class MojTyp>
```

a nagłówek klasy uogólniamy do:

```
Klasa<Typ>::Metoda (...)
```

⊙ Również wywołując konstruktor
lub podając typ obiektu naszej klasy
(która stała się szablonem) , piszemy:

```
Klasa<Typ>
```

[Link]

```
4  template <class T>
5  struct MyData {
6    T x;
7    MyData (T );
8    T    GetX ()     { return x; }
9    void SetX (T );
10   MyData operator+ (MyData& ) ;
11 };
12
13 template <class T>
14 MyData<T>::MyData (T _x) : x (_x) { }
15
16 template <class T>
17 void MyData<T>::SetX (T _x)  {
18   x = _x ;                    }
19
20 template <class T>
21 MyData<T> MyData<T>::operator+ (MyData<T>& Obj) {
22   return MyData<T> ( x + Obj.x ) ;
23 }
24
25 int main() {
26   MyData<int> iA ( 5 ) , iB ( -3 ) ;
27   cout << (iA + iB).GetX() << endl;
28 }
```

template class c. d.

[Link]

- Szablon może pracować na wielu typach. Wyszczególniamy je w <> po przecinku:

```
template<class Typ1, class Typ2>
```

Wówczas obiekt deklarujemy tak:

```
Klasa<Typ1, Typ2> Obiekt ;
```

- Możemy też podać domyślny typ, np.:

```
template<class U, class V = int>
```

Wówczas obiekt można zadeklarować bez specyfikowania tego typu. Taka deklaracja:

```
Klasa<Typ1> Obiekt ;
```

oznacza więc, że typem `V` jest `int`.

```
4  template <class U, class V = int>
5  struct MyPair {
6    U x;
7    V y;
8    MyPair (U _x, V _y) : x (_x) , y (_y) { }
9    void Print () ;
10 };
11
12 template <class U, class V>
13 void MyPair<U,V> :: Print () {
14   cout << "[ " << x << " : " << y << " ]\n";
15 }
16
17 int main() {
18   MyPair<int , char > P1 ( 12   , 'a' ) ;
19   MyPair<char, float> P2 ( 't' , 1.5 ) ;
20   MyPair<string>      P3 ("abc", 123 ) ;
21   P1.Print ();
22   P2.Print ();
23   P3.Print ();
24 }
```

template class c. d.

Szablonowi można podawać parametry pozatypowe (non-type). Np. gdy zapiszemy:

```
template<class Typ1, int len>
```

to obiekt można zadeklarować np. tak:

```
Klasa<Typ1, 4> Obiekt ;
```

Słowo „pozatypowy” może nieco mylić. Spośród *dopuszczonych możliwości*, warto wymienić `int` oraz `float`.

Ta funkcjonalność bardzo się przydaje, gdy szablon zawiera tablicę danych o elastycznym rozmiarze lub wymiarze.

[Link]

```
4  template <class T, int len>
5  struct MyData {
6    T Data[len] = {};
7    int size = len;
8    T& operator[] (int i) {
9      return Data[i];
10   }
11 };
12
13 int main() {
14   MyData< int , 8> iM ;
15   MyData<float, 4> fM ;
16   for (int i = 1; i <= iM.size; i++)
17     iM[i-1] = i;
18   for (int i = 1; i <= fM.size; i++)
19     fM[i-1] = 1. / (float) i;
20
21   for (auto& e : iM.Data) cout << e << ' ';
22   cout << endl;
23   for (auto& e : fM.Data) cout << e << ' ';
24   cout << endl;
25 }
```

- **template class c. d.**

⊙ Uogólnienie zaprzyjaźnienia (`friend`) w naszym szablonie dla zewnętrznej funkcji, która wykorzystuje nasz szablon.

Polecenie `friend` trzeba uogólnić. Ale podając tutaj typ, musimy dać inny, niż typ naszego szablonu. Powodem jest zakaz shadowingu zagnieżdżających się szablonów.

Np. gdy funkcja przyjmuje nasz szablon, To polecenie `friend` ma postać:

```
template<class U>
friend Typ funkcja (Klasa<U> Obj) ;
```

gdzie typ `U` jest inny niż `T`.

[Link]

```
4  template <class T>
5  struct MyData {
6    T x;
7    MyData (T );
8
9    template <class U>
10   friend MyData<U> operator* ( MyData<U>& Obj ) ;
11 };
12
13 template <class T>
14 MyData<T>::MyData (T _x) : x (_x) { }
15
16 template <class T>
17 MyData<T> operator* (T k, MyData<T>& Obj ) {
18   return MyData<T> ( k * Obj.x ) ;
19 }
20
21 int main() {
22   MyData<double> fA ( 3. ) ;
23   cout << ( 1.5 * fA ).x << endl;
24 }
```

- **template class c.d.** Jak podzielić kod z szablonem klasy?

⊙ Wykonujemy podstawowe kroki podziału (por. Wykład 8).

Przed końcem **pliku nagłówkowego** (.h) załączamy jego **plik implementacyjny** (.C) poleceniem:

```
#include "MyTemplate.C"
```

⊙ Możemy sprawdzać poprawność kodu szablonu poprzez:

```
g++ -fsyntax-only MyTemplate.h
```

⊙ Na końcu kompilujemy wyłącznie **plik klienta**:

```
g++ PlikKlienta.C -o aplikacja.exe
```

[Link]

```
#ifndef __MyData__
#define __MyData__

template <class T> struct MyData {
  T x;
  MyData (T );
  T    GetX ()     { return x; }
  void SetX (T );
  MyData operator+ (MyData& ) ;
};

#include "MyData.C"

#endif
```

```
template <class T>
MyData<T>::MyData (T _x) : x (_x) { }

template <class T>
void MyData<T>::SetX (T _x)  {
  x = _x ;                   }

template <class T>
MyData<T> MyData<T>::operator+
(MyData<T>& Obj) {
  return MyData<T> ( x + Obj.x ) ;
}
```

```
#include "MyData.h"
#include <iostream>
using namespace std;

int main( ) {
  MyData<int> iA ( 5 ) , iB ( -3 ) ;
  cout << (iA + iB).GetX( ) << endl;
}
```

- **pair <T1,T2>**
  to szablon do przechowywania par,
  które mogą być różnych typów.
  Jest dostępny w bibliotece <utility> .

⊙ Deklaracja pary z wpisaniem elementów:

```
pair<Typ1,Typ2>  P  (Obj1, Obj2);
```

⊙ Polecenie **make_pair** (Obj1, Obj2)
tworzy parę w tym miejscu.
Parę można też zwracać z funkcji.
Można też zwrócić przez zbiór: { .. , .. } .

⊙ Elementy pary P dostępne są przez:
P.first i **P.second**

⊙ Z pary można też rozpakować do zmiennych:
tie(Var1, Var2) = P ;

⊙ Od C++17 dostępne jest tzw.
structured binding (**wiązanie strukturalne**):

```
auto& [Var1, Var2] = P;
```

Zmienne są tu od razu deklarowane.
Można zablokować zapis: auto& → auto .

[Link]

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

pair<float,float>  MinMax (float* Data, int size)  {
  float Low = Data[0] , High = Data[0] ;
  for (int i = 0; i < size; i++) {
    Low  = min (Low , Data[i]) ;
    High = max (High, Data[i]) ;
  }
  return make_pair (Low, High) ;
//return { Low , High } ;          // Tak też można
}

int main () {
 float Data[7]= {4.2, 2.9, 5.8, 3.5, 4.7, 5.1, 2.1};

 pair<float,float> P = MinMax (Data, 7);
 cout << "Lo/Hi: " << P.first <<' '<< P.second << endl;

 float Lo, Hi;
 tie (Lo, Hi) = P ;
 cout << "Lo/Hi: " << Lo  <<' '<< Hi  << endl;

 auto [ Lo2 , Hi2 ] = P ;
 cout << "Lo/Hi: " << Lo2 <<' '<< Hi2 << endl;
}
```

[Link]

- **`tuple <T1,T2,...>`** [pl: ***krotka***]
  to szablon do przechowywania wielu obiektów, które mogą być różnych typów.
  Jest dostępny w bibliotece `<tuple>` .

⊙ Deklaracja tupli z wpisaniem elementów:

```
tuple<T1,T2,...> T = {01, 02, ...} ;
```

⊙ Polecenie **`make_tuple`** `(01,02,..)`
stworzy krotkę (tuplę) w danym miejscu.
Tuplę można też zwracać z funkcji.
Można też zwrócić przez zbiór: { .. , .. } .

⊙ Elementy tupli `T` dostępne są przez:
`get``<0>(T) , get<1>(T) , ...`

⊙ Z tupli można też rozpakować do zmiennych:
`tie` `(V1,V2,...) = T ;`

⊙ Od C++17 można też używać
structured binding (**wiązanie strukturalne**):
`auto& [V1,V2,...] = T;`

Zmienne są tu od razu deklarowane.
Można zablokować zapis: `auto&` → `auto` .

```
6  tuple<char,char,int> MinMaxSize (char* Napis)  {
7    char Lo = Napis[0], Hi = Napis[0];
8    int ind = 0;
9    for ( ; Napis[ind] != 0; ind++) {
10     Lo = min (Lo , Napis[ind]) ;
11     Hi = max (Hi , Napis[ind]) ;
12   }
13   return make_tuple (Lo, Hi, ind) ;
14 //return { Lo , Hi , ind } ;      // tak też można
15 }
16
17 int main () {
18  char Napis[] = "programowanie" ;
19
20  tuple<char,char,int> Info = MinMaxSize (Napis);
21  cout << "Min/Max/size: "  << get<0>(Info) <<' '<<
22          get<1>(Info) <<' '<< get<2>(Info) << endl;
23
24  char lo, hi;
25  int size;
26  tie (lo, hi, size) = Info;
27  cout << "Min/Max/size: " << lo <<' '<<
28          hi <<' '<< size << endl;
29
30  auto [ lo2 , hi2 , size2 ] = Info;
31  cout << "Min/Max/size: " << lo <<' '<<
32       hi <<' '<< size << endl;
33 }
```

- **`valarray <T>`**
  to szablon dla tablicy elementów typu T.
  Jest zaprojektowany pod działania blokowe.

⊙ Obiekt można deklarować z przypisaniem,
albo podając: tylko rozmiar
lub: inicjowaną wartość (jednolicie
na każdym z elementów) + rozmiar.

⊙ Metoda `size()` podaje liczbę elementów.
`Obiekt[i]` udostępnia i-ty element.
Metoda `sum()` zwraca sumę z elementów.

⊙ **Działania blokowe**
- ▻ operatory 2-argumentowe na obiektach
- ▻ funkcje analityczne (*spis funkcji*)
- ▻ własna funkcja (nast. strona)

⊙ Pętla zakresowa działa na `valarray` .

[Link]

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

int main () {
  valarray<float> arr1 = {1, 3, 5, 7, 9, 11};
  valarray<char>  arr2 (90);
  valarray<float> arr3 (-1, 6);

  cout <<    "size of arr1    : " << arr1.size ()
       << "\t Element arr1[3] : " << arr1[3]
       << "\t Sum of elements : " << arr1.sum() << endl;

  valarray<float> arr4 = ( arr1 + arr3 ) / arr3 ;

  for (auto& e : arr4)  cout << e << ' ' ;
  cout << endl;

  valarray<float> v_angles (90) , v_TrigOne (90);
  for (int i = 0; i < 90; i++)
    v_angles[i] = i * M_PI/180. ;

  v_TrigOne = sqrt ( pow( sin(v_angles) , 2.)
                   + pow( cos(v_angles) , 2.) );

  for (auto& e : v_TrigOne)  cout << e << ' ';
}
```

valarray <T> c.d.

- Obiekt `valarray` przekazany do funkcji zna swój rozmiar.
- Obiekty `valarray` są **sortowalne**. Dwoma pierwszymi argumentami `sort` są „adresy” pierwszego i ostatniego elementu do posortowania. Dokładniej – tzw. **iteratory** wskazujące na te elementy. Dla `valarray`, otrzymujemy je dzięki poleceniom `begin (..)` i `end (..)` .
- Metoda `apply(funkcja)` pozwala wywołać naszą funkcję na każdym elemencie.
- Polecenie `slice (begin, size, step)` zwraca podtablicę z elementów od `begin`, co krok `step` i rozmiaru `size`.
- Metoda `resize(x)` zmienia rozmiar obiektu. Niestety, zawartość się zeruje.

[Link]

```
7  void MyPrint (valarray<float>& V) {
8    for (auto& e : V)  cout << e << ' ';
9    cout << endl;
10 }
11
12 float MySquareFun (float x) {
13   return x*x ;
14 }
15
16 int main () {
17   valarray<float> vx = {6, 5, 4, 3, 2, 1, 0};
18
19   sort( begin(vx) , end(vx) ) ;
20   MyPrint ( vx );
21
22   valarray<float> vy = vx.apply ( MySquareFun ) ;
23   MyPrint ( vy );
24
25   valarray<float> vslice = vx [ slice (0, 5, 2) ];
26   MyPrint ( vslice ) ;
27
28   vx.resize (5);
29   MyPrint ( vx ) ;
30 }
```

- **Iterator** - to każdy obiekt, który wskazuje na element tablicy (**kontenera danych**). Jest to uogólnienie wskaźnika.

  `begin(T)` i `end(T)` zwracają iteratory: pierwszego elementu oraz po-ostatniego. Działają dla: tablic, `valarray`, ...

- W **<numeric>** i **<algorithm>** są polecenia do inicjowania i prostych działań. Polecenie:

▻ **iota** `( it0, it1, Base )`

przedział tablicy [it0, it1) wypełni liczbami [Base, Base+1, …) . Aby krok był inny, niż 1:

generate `(it0, it1, funkcja)`

▻ Iteratory min. / max. elementu tablicy zwrócą:

min_element / max_element `(it0, it1)`

▻ **S = accumulate** `(it0, it1, Base)`

z przedziału [it0, it1) policzy S = Base + Σ K_i

▻ **IP = inner_product** `(K0, K1, L0, Base)`

z tablic K i L (w przedziałach [K0, K1) i odpowiednio [L0,…)] policzy: IP = Base + Σ K_i L_i

[Link]

```
6  void Print (auto& Object)                    {
7    for (auto& e : Object) cout << e <<'\t';
8    cout << endl;                              }
9
10 int main() {
11   int T[4];
12   valarray<int> VA (4) , VB (4), VC (4);
13   int sum = 0, innprod = 0;
14
15   iota (  T      , T + 4   ,  1 );  Print (T );
16   iota (begin(VA), end(VA), -4 );  Print (VA);
17
18   generate (begin(VB), end(VB),
19     []() { static int i = 3; i += 2; return i; }
20   );
21   Print (VB);
22
23   cout << *min_element ( T, T+4 ) << ' '
24        << *max_element (begin(VA) , end(VA)) << endl;
25
26   cout << accumulate    (T, T+4, sum ) << endl;
27   cout << inner_product (T, T+4, begin(VA), innprod);
28 }
```

- Poeksplorujmy bibliotekę **<algorithm>** . Rozważmy przedział tablicy wskazywany przez iteratory [it0, it1) .

▻ Przyjrzyjmy się wpierw wypełnianiu:

`generate ( it0, it1, generator )`

Stronę temu widzieliśmy w roli `generator`'a – Lambdę. Teraz użyjemy zwykłej funkcji.

▻ `it find ( it0, it1, w )`

szuka wartości `w` i zwraca iterator tego elem.

▻ `copy ( it0, it1, itNew )`

kopiuje zakres do innej tabeli od poz. `itNew` .
Uwaga: docelowe miejsca muszą istnieć.

▻ `for_each (it0, it1, fun)`

dla każdego elementu wykonuje `fun` .

▻ `transform (it0, it1, itNew, fun )`

jak `for_each`, ale wynik wstawia do nowej tabeli od pozycji `itNew`.
Uwaga: docelowe miejsca muszą istnieć.

[Link]

```
7  float funGen ()                                  {
8    static float x = -1; x += 2; return x; }
9
10 void  sqr1 (float& x) { x *= 2. ;     }
11 float sqr2 (float  x) { return x*x ; }
12
13 void print (auto& Object)                        {
14   for (auto& e : Object) cout << e <<' ';
15   cout << endl;                                   }
16
17 int main () {
18   float T1[5], T2[5], T3[5];
19
20   generate ( begin(T1) , end(T1) , funGen );
21   print (T1);
22   float* pos = find ( begin(T1), end(T1) , 5. );
23   cout << "Position: " << pos - T1 << endl;
24
25   copy ( begin(T1) , end(T1) , begin(T2) ) ;
26   print (T2);
27   for_each  ( begin(T2), end(T2), sqr1 ) ;
28   print (T2);
29   transform (begin(T2), end(T2), begin(T3), sqr2);
30   print (T3);
31 }
```

- **<algorithm>** : **permutacje** i **przetasowania**

⊙ W zbiorze N elementów jest N! permutacji. Możemy modyfikować kontener do następnej lub poprzedniej permutacji jego elementów.

Dla przedziału konternera objętego [it0, it1) :

`next_permutation ( it0, it1 )`

zmodyfikuje kontener do następnej permutacji

`prev_permutation ( it0, it1 )`

cofnie kontener do poprzedniej permutacji.

⊙ Można też kontener przetasować losowo:

`random_shuffle ( it0, it1 )`

[Link]

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

void Print (auto& Tab)                          {
  for (auto& e : Tab) cout << e << ' ';
  cout << endl;                                 }

int main () {
  double T1[] = {1, 2, 3, 4, 5}; Print(T1);
  cout << endl;

  for (int i = 1; i <= 4; i++)                  {
    next_permutation (begin(T1) , end(T1));
    Print (T1) ;                                }
  cout << endl;

  for (int i = 1; i <= 4; i++)                  {
    prev_permutation (begin(T1) , end(T1));
    Print (T1) ;                                }
  cout << endl;

  for (int i = 1; i <= 4; i++)                  {
    random_shuffle (begin(T1) , end(T1));
    Print (T1) ;                                }
}
```