

WYDZIAŁ FIZYKI
UNIWERSYTET WARSZAWSKI

# Programowanie zaawansowane FM i NI

## Wykład 10

# Kontenery STL, iteratory, algorithm

Krzysztof Piasecki

Semestr letni roku akad. 2025-26

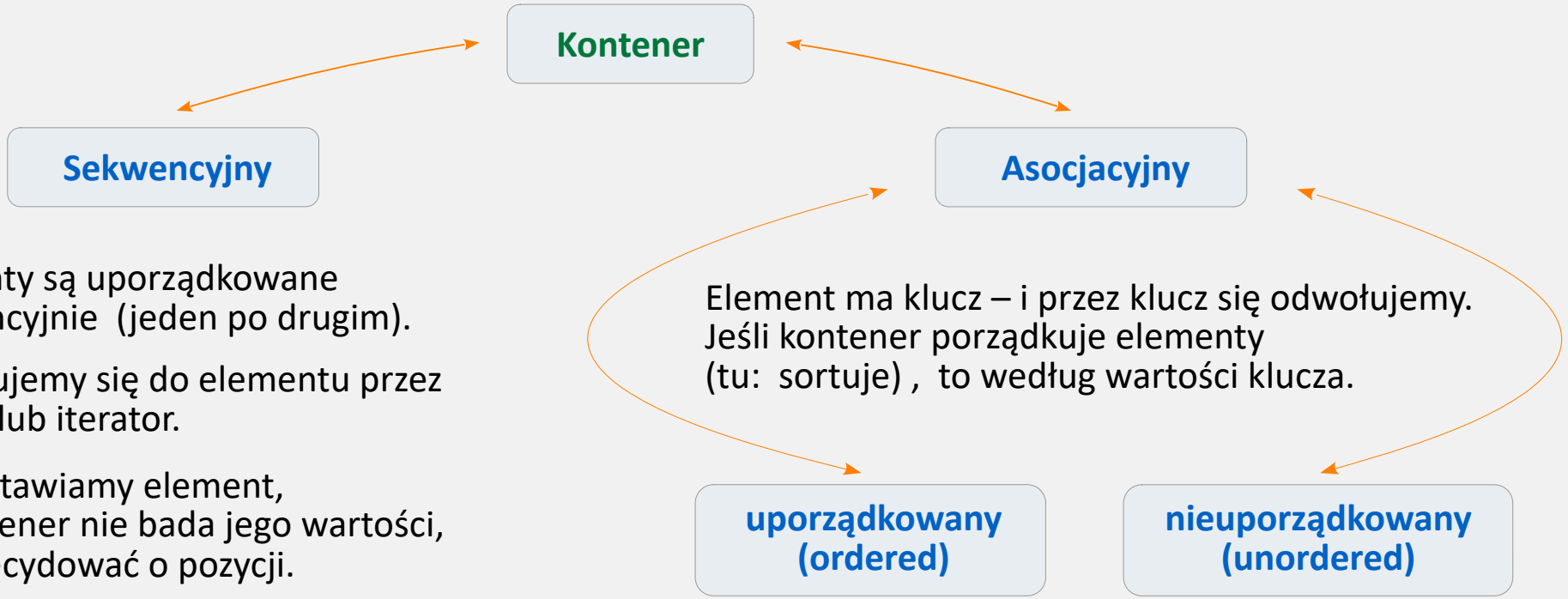

krzysztof.piasecki@fuw.edu.pl

- **STL (Standard Tempate Library)** – to zestaw szablonów, który dostarcza:
  **kontenery**, **iteratory** i **algorytmy**
  do operowania na różnych „typowych” kolekcjach danych.

**Kontener** (**container**) : to pojemnik o konkretnej budowie do przechowywania kolekcji danych jakiegoś typu.

*{**Adapter**: to pojemnik zbudowany na istniejącym kontenerze i zawężający jego budowę w danym celu.}*

**Iterator** : to obiekt wskazujący na jakiś element kontenera. Dzięki niemu można przebiegać po elementach.

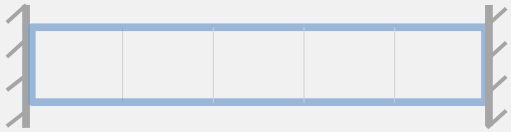



- Elementy są uporządkowane sekwencyjnie (jeden po drugim).
- Odwołujemy się do elementu przez indeks lub iterator.
- Gdy wstawiamy element, to kontener nie bada jego wartości, aby zdecydować o pozycji.

- **STL:** rodzaj kontenera do przechowania danych – zależy od naszej potrzeby.

**Kontenery sekwencyjne**. Wstawiając element, użytkownik decyduje o pozycji (w ramach możliwości).

**array**: tablica danych o zafiksowanej długości. Ciągła w pamięci.



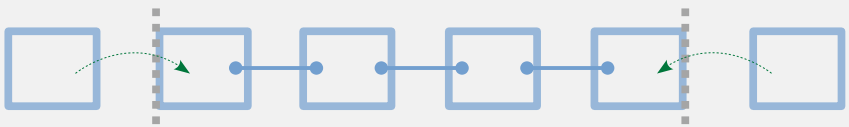
**vector**: tablica danych o długości elastycznej na końcu. Ciągła w pamięci.



**deque:** (*double-ended queue*):
tablica danych o długości elastycznej na początku i końcu. Nieciągła w pamięci.



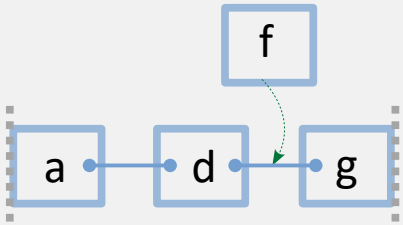
**list**: uporządkowana lista (brak indeksu, element zna sąsiadów). Nieciągła w pamięci.



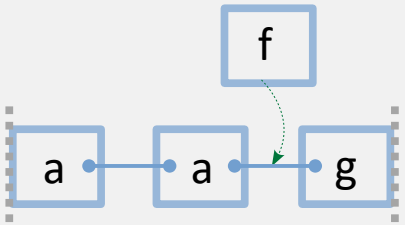
… i inne ...

- **Kontenery asocjacyjne**. Każdy element to albo „klucz”, albo para {klucz, wartość}. Nieciągłe w pamięci.

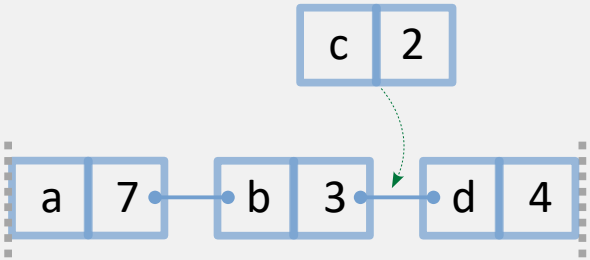
**set**: Posortowany zbiór kluczy.
Klucze są unikalne (każdy występuje max. 1 ×) .



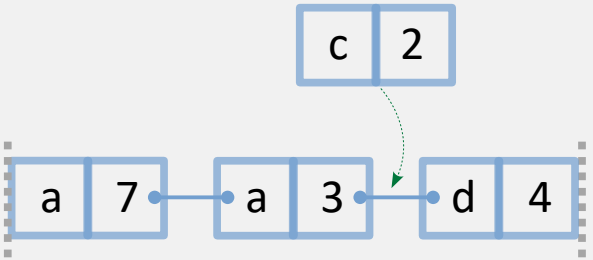

**multiset**: Posortowany zbiór kluczy.
Klucze nie muszą być unikalne.




**map**: Zbiór par {klucz, wartość} ,
posortowany wg kluczy.
Klucze są unikalne.




**multimap**: Zbiór par {klucz, wartość} ,
posortowany wg kluczy.
Klucze nie muszą być unikalne.




⊙ Dostępne są też warianty powyższych kontenerów, gdzie komputer nie sortuje elementów:

**unordered_set , unordered_multiset , unordered_map , unordered_multimap**

## Tworzenie kontenera

- Kod musi mieć **załączoną** odpowiednią **bibliotekę**, np. #include <vector>
  #include <map>

  *(nb. każdy multi….. ∈ do biblioteki bez multi w nazwie)*

- W **deklaracji** zawsze **konkretyzujemy typ(y)**. Dla `array`, dodajemy też rozmiar. Np.:

```
vector<float>        V ;
map<string, double>  M ;
array<int, 4>        A ;
```

- **Deklarując**, zawsze możemy **przypisać zbiór** (o adekwatnym typie) albo **obiekt** tego samego typu. Np:

```
deque<double>  D  =  { 1.2, 3.4, -5.6 } ;
deque<double>  D  (  { 1.2, 3.4, -5.6 }  ) ;
map<int,char>  M  =  {  {0, 'r'}  ,  {1, 'g'}  ,  {2, 'b'}  } ;
list<float> L  =  L2 ;
list<float> L  (  L2  ) ;
```

- Ponadto, `vector` i `deque` mają konstruktor wypełniający każdy element wartością:

```
vector<float> V ( 5, -12.34 ) ;
```
← 5 elementów, w każdym wartość -12.34

- Niemal każdy kontener **zna** swój **rozmiar**: `C.size();`

☞ Ten link ukazuje tabelę z zestawieniem metod i operatorów dla poszczególnych kontenerów.

- **Dostęp do elementu**

⊙ W każdym kontenerze sekwencyjnym mamy dostęp do **pierwszego elementu**: `C.front() ;`
W prawie każdym – dostęp do **ostatniego**: `C.back () ;`

⊙ W `array`, `vector` i `deque` mamy dostęp przez **numer indeksu**: `C[i]` lub `C.at(i) ;`

W `map` i `unordered_map` mamy dostęp przez **klucz**: `M["Jan"]` lub `C.at ("Ela") ;`

Przy czym `at(i)` sprawdza, czy indeks mieści się w zakresie (dla mapy: czy klucz istnieje) ,
a operator `[i]` tego nie sprawdza.

⊙ W *elastycznych* kontenerach sekwencyjnych **dokładanie** elementu na **koniec** (ew. **początek**) wygląda tak:

```
vector<int> V ;     V.push_back ( 12 );
deque <int> D ;     D.push_back ( 34 );      D.push_front ( 56 );
list  <int> L ;     L.push_back ( 78 );      L.push_front ( 90 );
```

⊙ a **kasowanie** elementu ostatniego (ew. początkowego) wygląda tak:

```
                    V.pop_back ();
                    D.pop_back ();           D.pop_front ();
                    L.pop_back ();           L.pop_front ();
```

- **Iterator**. Dla każdego rodzaju kontenera dostępny jest adekwatny iterator.
  To obiekt, który – jak wskaźnik – potrafi wskazywać na dany element.

⊙ Dla kontenera `K<typ>` dedykowany iterator będzie miał typ **`K<typ>::iterator`** . Czyli deklarujemy tak:

```
vector<float>::iterator       itV ;
map<string,double>::iterator  itM ;
```

⊙ Każdy kontener `K` poprzez metodę **`K.begin()`** zwraca iterator na pierwszy element,
a poprzez metodę **`K.end ()`** zwraca iterator na „po-ostatni” element. Np.:

```
array<int,4>::iterator   itA_pocz = A.begin()  ,  itA_konc = A.end() ;
```

⊙ Dla każdego z omawianych tu kontenerów, na jego iteratorze `it` można wykonać:

```
it++       ;     ++it      ;
it--       ;     --it      ;

*it        ;     it->a     ;

it1 == it2 ;   it1 != it2  ;
```

⊙ `array, vector i deque` potrafią też:

```
it += n  ;   it + n   ;   it1 > it2   ;   it[n]
it -= n  ;   it - n   ;   it1 < it2
```

[Tabela rodzajów iteratorów]

- **Pętla po kontenerze z użyciem iteratora**. Działa dla każdego kontenera. Np. dla `vector` :

```
vector<int> V  =  { 1, 2, 3, 4 } ;
vector<int>::iterator  it ;
for ( it = V.begin()  ;  it != V.end()  ;  it++ )
    cout <<  *it  << endl;
```

a dla iterowania po `map` (elementami są pary) , wyświetlanie każdego elementu wyglądałoby:

```
    cout <<  it->first  << ' ' <<  it->second  << endl;
```

- **Pętla zakresowa po kontenerze**. Też działa dla każdego kontenera. Np. dla `array` :

```
array<int,4>  A  = { 1, 2, 3, 4 };
for (auto& a : A)
  cout << ++a << endl ;
```

a dla iterowania po `map` , krok wyglądałby :

```
    cout << a.first << ' ' << a.second << endl;
```

- Od standardu C++17 działa też pętla przez **structured binding** (**wiązanie strukturalne**) :

```
map<int,char> M = { {0, 'r'} , {1, 'g'} , {2, 'b'} } ;
for ( auto& [key, value] : M )
  cout << key << ' ' << value << endl;
```

## Proste metody na kontenerach

- Każdemu kontenerowi `K` (oprócz `array`) można wpisać zbiór tych samych wartości:

  `K.assign (rozmiar, wartość);`

- Każdy kontener `K1` można zamienić z `K2` :

  `K1.swap ( K2 );`

- W każdym kontenerze (oprócz `array`) można wtrącić element, podając pozycję:

  `K1.insert ( K1.begin()+2 , -34 );`

- oraz skasować element:

  `K1.erase ( K1.begin()+2 );`

- Każdemu kontenerowi (oprócz `array`) można zmienić rozmiar. Elementy mieszczące się w rozmiarze nie będą kasowane.

  `K1.resize ( 5 );`

- W każdym asocjacyjnym kontenerze można policzyć wystąpienia elementu:

  `S.count ( 3 );`

```cpp
void print ( const string& txt, vector<auto>& V ) {
  cout << "[" << txt << " | " ;
  for (auto& el : V)  cout << el << ' ' ;
  cout << "]" << endl;
}

int main () {
  vector<int> v1 = {1, 2, 3} , v2 ;
  v2.assign ( 4, -12 );
  print ("v1", v1 ) ;  print ("v2", v1 );

  v1.swap ( v2 ) ;
  print ("v1", v1 ) ;  print ("v2", v2 ) ;

  v2.insert ( v2.begin() + 1 , 0 );
  print ("v2", v2 ) ;

  v2.erase  ( v2.end() - 1 ) ;
  print ("v2", v2 ) ;

  v2.resize ( 6 ) ;
  print ("v1", v1 ) ;

  multiset<int> S = { 1, 2, 3, 3, 2 };
  cout << "3 appears " << S.count ( 3 ) << " times.";
}
```

[Link]

## Kontener zawierający kontenery

- Ważny przykład: **`vector< vector<Typ> >`** . Dzięki temu można otrzymać macierz. Np.:

```
vector< vector<int> > W = { {1, 2, 3 },
                            {4, 5, 6 },
                            {7, 8, 9 } };
```

*Uwaga*: elementami kontenera W są obiekty `vector<int>`. Każdy opisuje „wiersz macierzy”. Jeśli chcemy wpisywać wiersze „po kolei” , to można np. tak:

```
vector<vector<int>> W ;
W.push_back ( {1, 2} );
W.push_back ( {3, 4} );
```

- Dostęp do komórek macierzy: albo `W[rząd][kolumna]`
  albo (bezpieczniej) `W.at(rząd).at(kolumna)`

- Co z **`valarray`**? Szablon `valarray` formalnie nie wszedł do STL. Mimo podobnej funkcjonalności, różni się technicznie. Np. nie posiada metod `begin()` i `end()` , zwracających iteratory. Ale można je wydobyć tak: `begin( v )` i `end( v )`.

- Eleganckie **przepisanie danych `valarray` ↔ `vector`** :

```
vector  <double> vecData ( begin (valData) , end (valData)  );
valarray<double> valNew  ( vecData.data()  , vecData.size() );
```

[Link]

- **Różne rodzaje map**

| | | |
|---|---|---|
| `map` | : | sortuje, klucze unikalne |
| `multimap` | : | sortuje, klucze wielokrotne |
| `unordered_map` | : | nie sortuje, klucze unikalne |
| `unordered_multimap` | : | nie sortuje, klucze wielokrotne |

⊙ Dla mapy `M` z kluczami unikalnymi, napisanie:

```
M[ klucz ] = wartość ;
```

jest wstawieniem pary, jeśli takiej nie było.
Jeśli była, to jest zmianą wartości tego klucza.

⊙ Wywołanie, np. takie:

```
cout << M[ klucz ] ;
```

zwraca wartość dla klucza, o ile klucz istnieje.
Ale gdyby nie istniał, to tworzy go, z wartością 0.

⊙ Metoda **`insert`** wstawia parę do mapy.
To „typowy” sposób dla map z kl. wielokrotnymi.
Ale jeśli klucze mapy mają być unikalne,
a dany klucz już jest, to `insert` nic nie zrobi.

```
6  void print (auto& M) {
7    for (auto& [key, value] : M)
8      cout << '[' << key << ' ' << value << "] ";
9    cout << endl;
10 }
11
12 int main () {
13   unordered_map <string, int> UM;
14   UM ["ed" ] = 56;
15   UM ["ala"] = 34;
16   UM ["jan"] = 12;
17   print ( UM ) ;
18   UM ["ala"] = 78;
19   UM.insert ( {"ala", 90} ) ;
20   print ( UM ) ;
21   cout << UM ["ufo"] << endl;
22   print ( UM ) ;
23
24   map      <string,int> M  ( UM.begin() , UM.end() );
25   print ( M ) ;
26
27   multimap<string,int> MM (  M.begin() ,  M.end() );
28   MM.insert ( {"ala", 78} ) ;
29   print ( MM );
30 }
```

- **Biblioteka <algorithm>**

[Link]

Na Wykładzie 9 pokazaliśmy działanie kilku funkcji w **<algorithm>** na tablicach statycznych i na `valarray`.

⊙ Tu widzimy działanie tych samych funkcji, ale na obiektach `vector` . (tylko iteratory zadajemy inaczej).

⊙ To samo zadziała dla innych sekwencyjnych kontenerów, czyli: `array` , `deque` , `list` , ...

⊙ Dla asocjacyjnych – trzeba sprawdzać.

Przykładowo:

`for_each` dla `map` zadziała, gdy w argumencie wołanej funkcji (typu `pair`) klucz będzie `const` :

```
void F(pair<const int,int>& P) {
 P.second *= -1;                }
...
map<int,int> M = { {1,2}, {3,4} };
for_each (M.begin(), M.end(), F);
```

```
7  void   sqr1 (double& x) { x *= 2;     }
8  double sqr2 (double  x) { return x*x; }
9
10 void print (vector<double>& V) {
11   for (auto& e : V) cout << e << ' ';
12   cout << endl;
13 }
14
15 int main () {
16   vector<double> V1 = {1, 2, 3, 4, 5} , V2(5) , V3(5);
17
18   cout << find (V1.begin(), V1.end(), 4. )
19                                - V1.begin() << endl;
20
21   copy ( V1.begin() , V1.end() , V2.begin() ) ;
22   print (V2);
23
24   for_each  (V2.begin(), V2.end(), sqr1 ) ;
25   print (V2);
26
27   transform (V2.begin(), V2.end(), V3.begin(), sqr2 );
28   print (V3);
29 }
```

- **<algorithm>** - poznajmy coś jeszcze.

⊙ Istnieją funkcje warunkowe :

**find_if count_if, copy_if replace_if, erase_if, remove_if**

Rozpatrujac każdy element, wywołują „**predykat**” (**predicate**) : to funkcja, która musi zwrócić `true` , aby zaszło działanie.

Przykładowo, dla:

```
deque<int> D = {1, 2, 3};
```

gdy mamy predykat `IsOdd` (zobacz tu) to zliczacz wystąpień: `count_if` (tu) zwróci 2, bo są 2 liczby nieparzyste.

⊙ *Uwaga*: `copy_if` ani `remove_if` nie skracają kontenera docelowego. `remove_if` : elementy spełniające predykat są tylko przesuwane na lewo, a funkcja zwraca iter. „nowego” końca.

Wersja z usuwaniem to: `erase_if`

```
 8  bool IsOdd  (int I) { return I % 2 != 0 ; }
 9
10  bool IsZero (int I) { return I == 0     ; }
11
12  void Print (auto& Cont)                    {
13    for (int& e : Cont)  cout << e << '\t';
14    cout << endl;                            }
15
16  int main () {
17    deque<int> D = { 1, 2, 3, 4, 5 } , D2 (5) ;
18    cout << * find_if (D.begin(), D.end(), IsOdd) << endl
19         << count_if (D.begin(), D.end(), IsOdd) << endl;
20
21    copy_if  ( D.begin() , D.end() , D2.begin() , IsOdd );
22    Print ( D2 ) ;
23
24    replace_if ( D.begin(), D.end(), IsOdd, 0 ) ;
25    Print ( D ) ;
26
27    remove_if  ( D.begin(), D.end(), IsZero ) ;
28    Print ( D ) ;
29
30    erase_if ( D , IsZero ) ;
31    Print ( D ) ;
32  }
```

**[Link]**

- **<algorithm>** - jeszcze o predykatach.

⊙ Dzięki predykatowi można pytać o:

**all_of ( it0, it1, predicate )**

czy wszystkie elementy go spełniają?

**any_of ( it0, it1, predicate )**

czy choć jeden element go spełnia?

**none_of ( it0, it1, predicate )**

czy żaden z elementów go nie spełnia.

Funkcje te zwracają `true / false`.

⊙ Nb. co może być predykatem?

- ▻ funkcja
- ▻ wyrażenie lambda
- ▻ funktor

Drogą lambdy i funktora można też porównać elementy z argumentem!

[Link]

```
6  bool IsOdd  (int I) { return I % 2 != 0 ; }
7
8  struct IsZero                                    {
9   bool operator() (int i) { return i == 0; }      };
10
11 struct Equals                                    {
12  int data;
13  Equals (int _data) : data (_data) { }
14  bool operator() (int i) { return i == data; }   };
15
16 int main () {
17  vector<int> V = {-1, 1, 3, 5, 7, 9};
18
19  cout << boolalpha; // will print bools as true/false
20
21  cout << all_of  (V.begin() , V.end() , IsOdd ) << endl;
22
23  cout << any_of  (V.begin() , V.end() ,
24                   [](int& x) { return x >= 0; } ) << endl;
25
26  cout << none_of (V.begin() , V.end() , IsZero()  ) << endl;
27
28  cout << none_of (V.begin() , V.end() , Equals(0) ) << endl;
29 }
```