



C++17 i STL

Iteratory



Iteratory

- ▶ Iterator to obiekt reprezentujący pozycję elementu w kontenerze.
- ▶ Zachowanie iteratora definiują podstawowe operacje:
 - ▶ Operator `*` zwraca element z aktualnej pozycji. Jeśli elementy posiadają składowe, dostęp do nich można uzyskać bezpośrednio z poziomu iteratora za pomocą operatora `->`.
 - ▶ Operator `++` pozwala iteratorowi przejść do następnego elementu. Większość iteratorów pozwala również na wykonanie kroku wstecz za pomocą operatora `--`.
 - ▶ Operatory `==` oraz `!=` zwracają wartość logiczną informującą, czy dwa iteratory reprezentują tę samą pozycję.
 - ▶ Operator `=` przypisuje iterator (pozycję elementu, do którego on się odnosi).
- ▶ Operatory te są interfejsem zwykłych wskaźników języka C++ wykorzystywanych do iterowania po elementach tablicy.



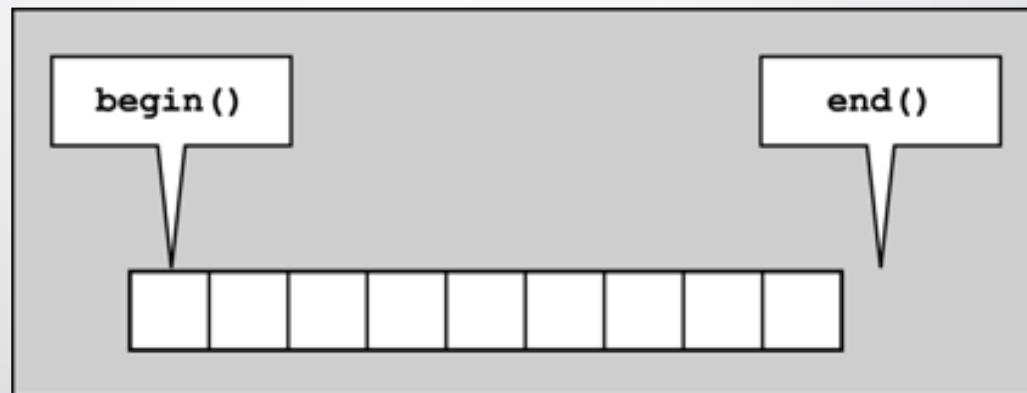
Iteratory



- ▶ Iteratory są inteligentnymi wskaźnikami, które iterują po bardziej skomplikowanych strukturach danych niż tablice.
- ▶ Wewnętrzne zachowanie iteratorów zależy od struktury danych, po której one iterują – każdy typ kontenerowy zapewnia swój własny rodzaj iteratora (iteratory współdzielą ten sam interfejs, lecz posiadają różne typy).
- ▶ Wszystkie klasy kontenerowe udostępniają te same podstawowe funkcje składowe, które umożliwiają im wykorzystanie iteratorów do nawigowania po ich elementach:
 - ▶ `begin()` – zwraca iterator reprezentujący początek elementów w kontenerze (początkiem tym jest pozycja pierwszego elementu, jeśli kontener nie jest pusty).
 - ▶ `end()` – zwraca iterator reprezentujący koniec elementów w kontenerze (końcem tym jest pozycja za ostatnim elementem).

Iteratory

- ▶ Funkcje `begin()` oraz `end()` definiują zakres półotwarty zawierający pierwszy element, lecz wyłączający ostatni.
- ▶ Zakres półotwarty posiada dwie zalety:
 - ▶ Istnieje proste kryterium zakończenia pętli iterującej po elementach — wykonywanie pętli kontynuowane jest tak długo, dopóki nie zostanie osiągnięta wartość funkcji `end()`.
 - ▶ Unikamy konieczności specjalnej obsługi zakresów pustych. Dla zakresów pustych wartość funkcji `begin()` równa jest wartości funkcji `end()`.






Iteratory

► Przykład:

```
list<char> coll; // kontener list na elementy znakowe
// ...
// wypisz wszystkie elementy - iteruj po elementach
list<char>::const_iterator pos;
for (pos = coll.cbegin(); pos != coll.cend(); ++pos)
    cout << *pos << ' ';
cout << endl;
```




Przesuwanie iteratora po kolekcji

- ▶ Do przesunięcia iteratora do następnego elementu można użyć prefikсового operatora inkrementacji (`++iter`) albo postfiksowego (`iter++`).
- ▶ Należy używać prefikсового operatora inkrementacji ponieważ może on mieć lepszą wydajność niż operator postfiksowy – ten ostatni wewnętrznie wykorzystuje obiekt tymczasowy, gdyż musi zwrócić starą pozycję iteratora.

Iteratory modyfikujące i niemodyfikujące

- ▶ Każdy kontener definiuje dwa typy iteratorów:
 - ▶ `kontener::iterator` przeznaczony do iterowania po elementach w trybie z odczytem i zapisem;
 - ▶ `kontener::const_iterator` przeznaczony do iterowania po elementach w trybie tylko do odczytu.
- ▶ Przykład:


```
list<char>::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
}
```



Dedukowanie typu iteratora za pomocą auto

- ▶ Można zainicjalizować iterator wywołaniem funkcji składowej `begin()` kontenera i dzięki temu pominąć jawną deklarację jego typu używając słowa `auto`.
- ▶ Przykład:

```
for (auto it = coll.begin(); it != coll.end(); ++it) {  
    cout << *it << ' ' ;  
}
```
- ▶ Zaletą stosowania `auto` jest zdecydowane polepszenie przejrzystości kodu.
- ▶ Wadą stosowania `auto` jest utrata jawnie deklarowanej niemodyfikowalności iteratora, ponieważ po zastosowaniu inicjalizacji za pomocą `coll.begin()` otrzymamy iterator modyfikujący (bez `const`).



Zachowanie pierwotnych wartości w kolekcji

- ▶ Wartość zwracana z funkcji składowej `begin()` jest obiektem typu *kontener::iterator*.
- ▶ Aby zachować możliwość stosowania iteratorów niemodyfikujących i równocześnie zapewnić wygodę stosowania `auto`, w klasach kontenerów w C++11 udostępniono funkcje składowe `cbegin()` i `cend()` – obie zwracają obiekt iteratora typu *kontener::const_iterator*.
- ▶ Iterator typu `const_iterator` nie pozwala na modyfikację elementów kolekcji.

Iteratory a pętle zakresowe

- ▶ W przypadku kontenerów zakresowa pętla for jest niczym innym jak wygodnym interfejsem, zdefiniowanym do iterowania po wszystkich elementach przekazanego zakresu kolekcji (od początku do końca).
- ▶ Wewnątrz ciała pętli bieżący element jest inicjalizowany wartością, do której odnosi się bieżący iterator.

- ▶ Przykład:

```
for (auto elem: coll) { ... }  
jest równoważna z  
for (auto it = coll.begin(); it != coll.end(); ++it) {  
    auto elem = *pos;  
    ...  
}
```

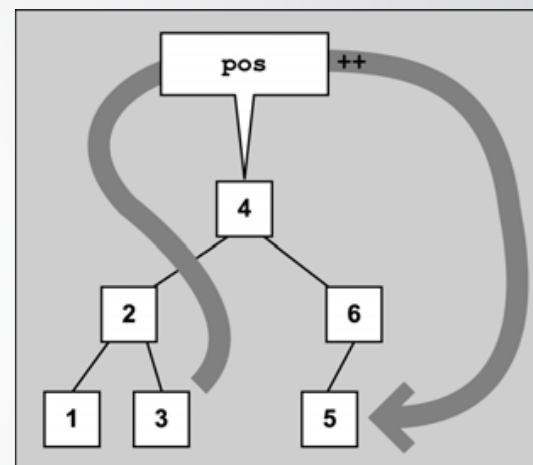
- ▶ W zakresowej pętli for warto deklarować elem jako niemodyfikującą referencję – w taki sposób eliminuje się konieczność tworzenia kopii elementów przeglądanej kolekcji.

- ▶ Przykład:

```
for (const auto &elem: coll) { ... }
```

Użycie iteratorów w kontenerach

- ▶ Ponieważ iterator zdefiniowany jest przez kontener, działa on prawidłowo nawet w przypadku, gdy wewnętrzna struktura kontenera jest bardziej skomplikowana.
- ▶ Jeśli na przykład iterator odnosi się do trzeciego elementu, operator ++ przenosi go do elementu czwartego znajdującego się na wierzchołku drzewa.
- ▶ Po następnym wywołaniu operatora ++ iterator będzie się odnosić do piątego elementu umieszczonego u spodu drzewa.

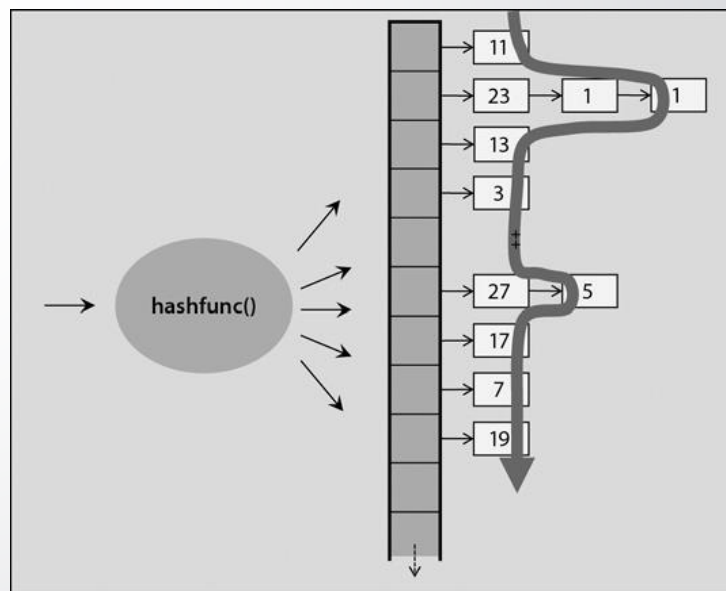


Użycie iteratorów w kontenerach – przykład 1

```
using IntSet = set<int, greater<int>>;  
...  
IntSet coll;  
...  
coll.insert({3, 1, 5, 4, 1, 6, 2});  
...  
IntSet::const_iterator pos;  
for (pos = coll.begin(); pos != coll.end(); ++pos)  
    cout << *pos << ' '  
cout << endl;
```

Użycie iteratorów w kontenerach

- ▶ Za pomocą iteratorów można przeglądać zawartość kontenerów nieporządkowanych. Kolejność elementów w takim kontenerze jest nieokreślona – zależy ona od wewnętrznego stanu tablicy i funkcji haszującej.



Użycie iteratorów w kontenerach – przykład 2

```
using IntUnMSet = unordered_multiset<int>;
...
IntUnMSet coll;
...
coll.insert({2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 2});
...
for (auto elem : coll)
    std::cout << elem << ' ';
cout << endl;
...
coll.insert(31);
...
for (auto elem : coll)
    std::cout << elem << ' ';
cout << endl;
```




Kategorie iteratorów

- ▶ Oprócz podstawowych funkcjonalności, iteratory mogą posiadać również zdolności zależne od wewnętrznej struktury kontenera.
- ▶ Biblioteka STL udostępnia tylko te funkcjonalności iteratorów, które oferują dobrą wydajność – jeśli na przykład kontenery posiadają dostęp swobodny, tak jak `vector<>` czy `deque<>`, to ich iteratory także potrafią wykonywać operacje dostępu swobodnego.
- ▶ Iteratory predefiniowanych klas kontenerowych należą do jednej z następujących trzech kategorii:
 - ▶ iterator postępujący – zdolny do iterowania tylko w przód;
 - ▶ iterator dwukierunkowy – zdolny do iterowania w dwóch kierunkach: w przód oraz wstecz;
 - ▶ iterator dostępu swobodnego – posiada wszystkie właściwości iteratora dwukierunkowego oraz dodatkowo potrafi realizować dostęp swobodny.
- ▶ Dodatkowo zdefiniowano dwie inne kategorie iteratorów:
 - ▶ iterator wejściowy – zdolny do odczytywania (przetwarzania) wartości podczas iterowania w przód;
 - ▶ iterator wyjściowy – zdolny do wypisywania wartości podczas iterowania w przód.


Iteratory strumieniowe

- ▶ Iterator strumieniowy (ang. stream iterator) to adaptator iteratora, który umożliwia wykorzystanie strumienia jako źródła lub przeznaczenia algorytmów.
- ▶ W szczególności iterator strumieniowy wejściowy służy do odczytywania elementów ze strumienia wejściowego, a iterator strumieniowy wyjściowy do wpisywania wartości do strumienia wyjściowego.
- ▶ Definicje iteratorów strumieniowych znajdziemy w pliku nagłówkowym <iterator>.



Iteratory strumieniowe – przykłady

```
istream_iterator<int> it_cin {cin};  
istream_iterator<int> end_cin;  
...  
deque<int> v;  
...  
copy(it_cin, end_cin, back_inserter(v));  
...  
for (auto elem : v)  
    cout << elem << ' '  
cout << endl;
```



Iteratory strumieniowe

– przykłady

```
istream_iterator<int> it_cin {cin};
istream_iterator<int> end_cin;
...
deque<int> v;
...
while (it_cin != end_cin) {
    v.push_back(*it_cin++);
}
...
for (auto elem : v)
    cout << elem << ' ';
cout << endl;
```