Kurs języka C++

12. Kolekcje

Spis treści

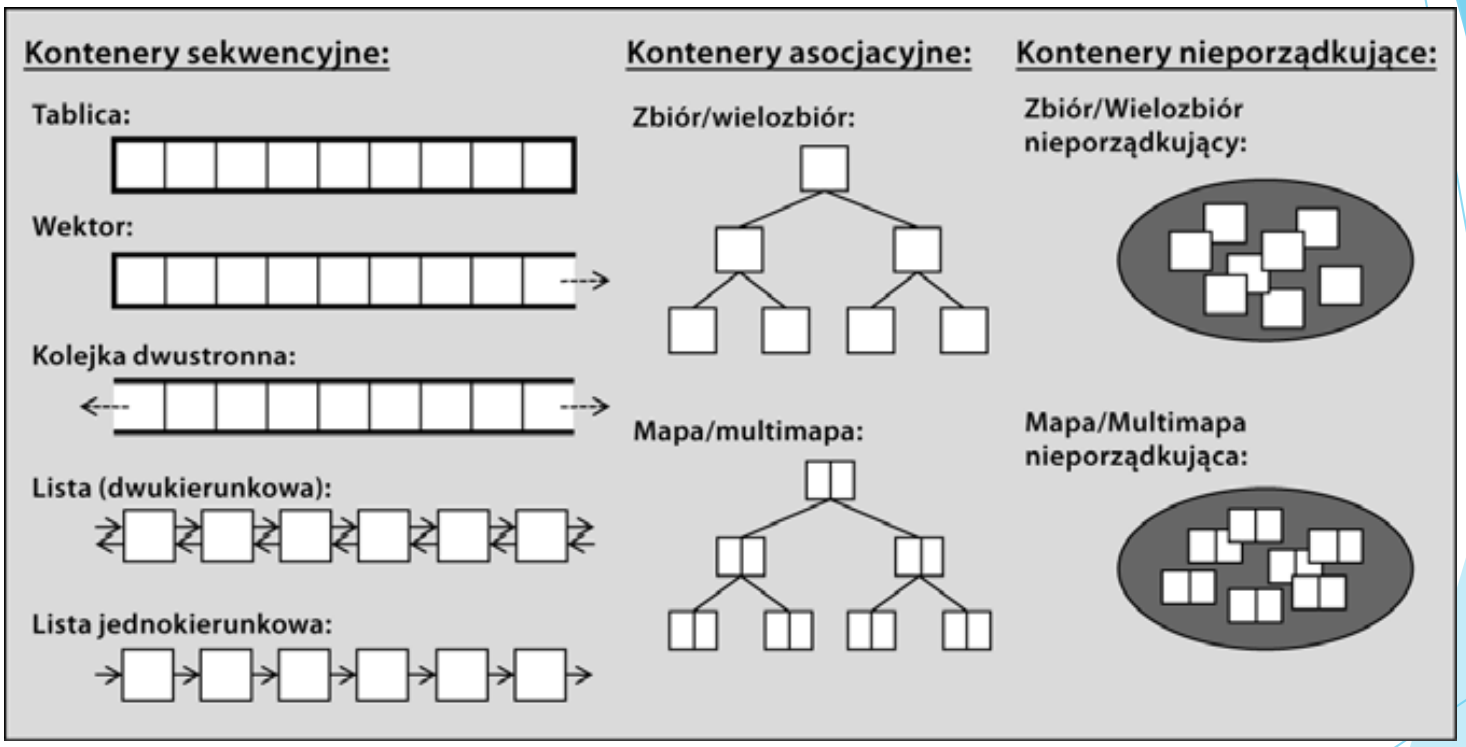
Kontenery

- Kontenery służą do przechowywania i zarządzania kolekcjami danych.
- Rodzaje kontenerów:
  - Kontenery sekwencyjne, gdzie każdy element ma określoną pozycję. Na przykład: `array`, `vector`, `deque`, `list`, `forward_list`. Kontenery sekwencyjne są zbudowane na tablicach dynamicznych albo na listach.
  - Kontenery uporządkowane (w tym asocjacyjne), gdzie pozycja elementu zależy od jego wartości. Na przykład: `set`, `multiset`, `map`, `multimap`. Kontenery asocjacyjne są zbudowane na zrównoważonych drzewach BST.
  - Kontenery nieuporządkowane, gdzie pozycja elementu nie zależy od jego wartości. Na przykład: `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`. Kontenery nieuporządkowane są zbudowane na tablicach z haszowaniem.

Kontenery

- Kontenery sekwencyjne będące kolekcjami uporządkowanymi, w których każdy element posiada określoną pozycję. Pozycja ta zależy od momentu oraz miejsca wstawienia, jest jednak niezależna od wartości elementu.
- Kontenery asocjacyjne będące kolekcjami sortowanymi, w których aktualna pozycja elementu zależy od jego wartości (albo klucza w przypadku kontenerów operujących na parach klucz-wartość), zgodnie z określonym kryterium sortowania.
- Kontenery asocjacyjne nieporządkujące to kolekcje nieporządkujące i niezachowujące pozycji elementów, bo ich zadaniem głównym jest ustalanie, czy (a nie gdzie) element znajduje się w kolekcji. Elementy nie zachowują więc uporządkowania ani względem kolejności wstawiania, ani względem wartości — jedno i drugie może w czasie życia kontenera ulegać zmianie.

Kontenery


Kontenery sekwencyjne:
Tablica:
Wektor:
Kolejka dwustronna:
Lista (dwukierunkowa):
Lista jednokierunkowa:
Kontenery asocjacyjne:
Zbiór/wielozbiór:
Mapa/multimapa:
Kontenery nieporządkujące:
Zbiór/Wielozbiór nieporządkujący:
Mapa/Multimapa nieporządkująca:

Elementy kontenerów

- Elementy kontenerów muszą spełniać wymagania podstawowe:
  - element musi być *kopiowalny* (konstruktor kopiujący),
  - element musi być *przypisywalny* (przypisanie kopiujące),
  - element musi być *zniszczalny* (publiczny destruktor).
- W pewnych sytuacjach elementy kontenerów muszą spełniać wymagania dodatkowe:
  - konstruktor domyślny (utworzenie niepustego kontenera),
  - operator porównywania == (wyszukiwanie),
  - operator porównywania < (kryterium sortowania).

Semantyka wartości a semantyka referencji

- Kontenery STL realizują semantykę wartości: tworzą wewnętrzne kopie swoich elementów oraz zwracają kopie tych elementów.
- Semantykę referencji można zaimplementować samodzielnie za pomocą inteligentnych wskaźników – wskaźniki te mają umożliwiać zliczanie referencji dla obiektów, do których odnoszą się wskaźniki.

Wspólne cechy kontenerów

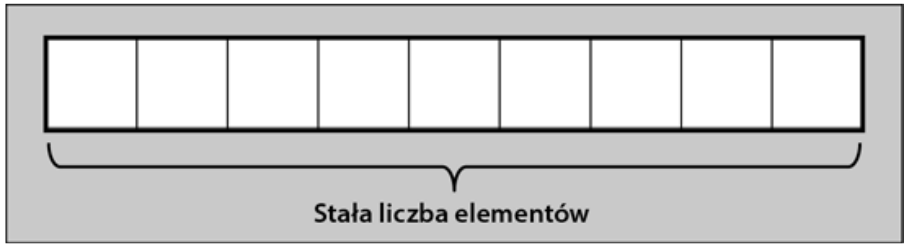
- Wszystkie kontenery zapewniają semantykę wartości.
- Wszystkie elementy posiadają określoną kolejność (kontenerowy udostępniają operacje zwracające iteratory służące do iteracji po kolejnych elementach).
- Operacje na kontenerach nie są bezpieczne, czyli nie sprawdzają możliwości wystąpienia każdego rodzaju błędu (to funkcja wywołująca musi zapewnić spełnienie wymagań przez parametry operacji).

Wspólne operacje na kontenerach

- Konstruktror domyślny, kopiujący, przenoszący, zakresowy (kopiuje elementy z innej kolekcji z podanego za pomocą iteratorów zakresu), z listą wartości.
- Destruktor (usuwa wszystkie wartości).
- Przypisanie kopiujące i przenoszące.
- Funkcje składowe `empty()`, `size()`, `clear()`, `swap(coll)`.
- Funkcja globalna `swap()`.
- Operatory relacyjne ==, !=, <, <=, >, >=.
- Iteratory `begin()`, `end()`, `cbegin()`, `cend()`.

Tablice

- Tablica to egzemplarz klasy kontenera `array<>`.
- Tablica modeluje tablicę statyczną (jest to otoczka dla statycznej tablicy z języka C, zapewniająca interfejs kontenera STL).
- Tablice kopiują elementy do własnych wewnętrznych, statycznych tablic.



Tablice

- Tablica `array<>` to jedyny kontener, którego elementy są inicjalizowane domyślnie, jeśli nic nie zostanie przekazane jawnie.
- Inicjalizacja tablic:

```
std::array<int,4> x;
// elementy x posiadają niezdefiniowane wartości
std::array<int,4> x = {};
// wszystkie elementy x mają wartość domyślną 0 (int())
```

Tablice - przykłady

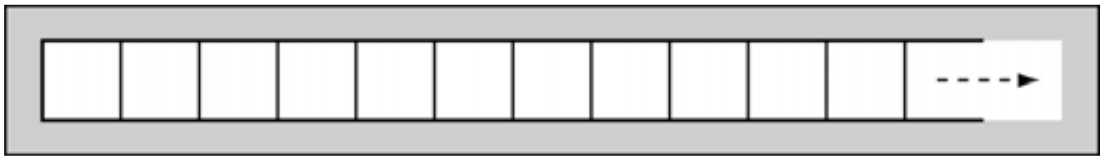
```
// tworzenie wykorzystujące inicjalizację zbiorczą
std::array<int, 3> a1{ {1, 2, 3} }; // podwójne klamry są wymagane
std::array<int, 3> a2 = {1, 2, 3};  // nie są potrzebne po znaku =
std::array<std::string, 2> a3 = { std::string("a"), "b" };

// Pozwala na wykonywanie operacji jak na zwykłym kontenerze
std::sort(a1.begin(), a1.end());
std::reverse_copy(
   a2.begin(), a2.end(),
   std::ostream_iterator<int>(std::cout, " "));
std::cout << endl;

// Pozwala na użycie zakresowej pętli for
for(const auto& s: a3)
   std::cout << s << ' ';
```

Kontenery sekwencyjne – wektory

- Wektor `vector<>` (zdefiniowany w `<vector>`) przechowuje swoje elementy w tablicy dynamicznej.
- Uzyskujemy szybki dostęp do każdego elementu za pomocą indeksowania.
- Dołączanie i usuwanie elementów na końcu wektora jest bardzo szybkie, ale wstawienie lub usunięcie elementu ze środka zabiera więcej czasu.



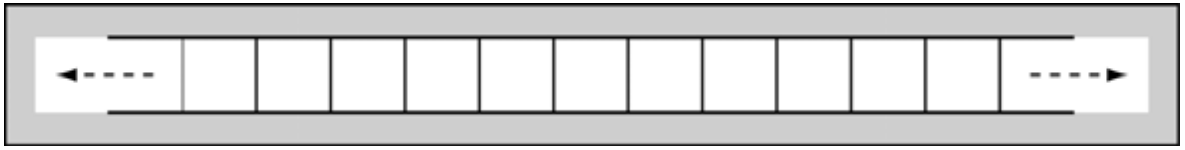
Kontenery sekwencyjne - wektory

- Przykład:

```
vector<int> coll;
…
for (int i=1; i<=6; ++i)
  coll.push_back(i);
…
for (int i=0; i<coll.size(); ++i)
  cout << coll[i] << ' ';
cout << endl;
```

Kontenery sekwencyjne – kolejki o dwóch końcach

- Kolejka o dwóch końcach `deque<>` (zdefiniowana w `<deque>`) przechowuje swoje elementy w tablicy dynamicznej, która może rosnąć w dwie strony.
- Uzyskujemy szybki dostęp do każdego elementu za pomocą indeksowania.
- Dołączanie i usuwanie elementów na końcu i na początku kolejki jest bardzo szybkie, ale wstawienie lub usunięcie elementu ze środka zabiera więcej czasu.



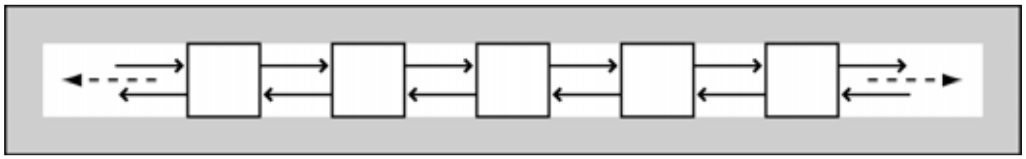
Kontenery sekwencyjne - kolejki o dwóch końcach

- Przykład:

```
deque<float> coll;
…
for (int i=1; i<=6; ++i)
  coll.push_front(i*1.234);
…
for (int i=0; i<coll.size(); ++i)
  cout << coll[i] << ' ';
cout << endl;
```

Kontenery sekwencyjne – listy

- Lista `list<>` (zdefiniowana w `<list>`) przechowuje swoje elementy w liście dwukierunkowej.
- W listach nie ma swobodnego dostępu do elementów kolekcji.
- Dołączanie i usuwanie elementów na końcu i na początku listy jest bardzo szybkie, ale dostanie się do elementu ze środka zabiera dużo czasu.



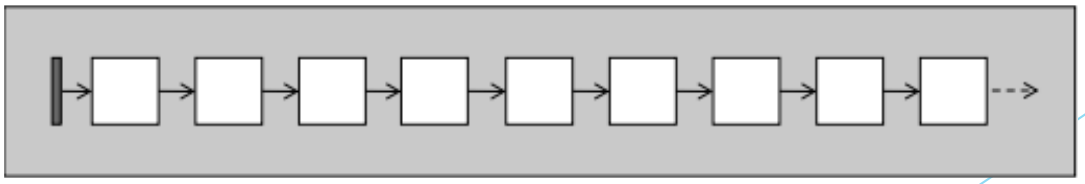
Kontenery sekwencyjne - listy

- Przykład:

```
list<char> coll;
…
for (char c='a'; c<='z'; ++c)
  coll.push_back(c);
…
while (!coll.empty()) {
  cout << coll.front() << ' ';
  coll.pop_front(); }
cout << endl;
```

Kontenery sekwencyjne – listy jednokierunkowe

- Lista `forward_list<>` (zdefiniowana w `<forward_list>`) przechowuje swoje elementy w liście jednokierunkowej.
- W listach nie ma swobodnego dostępu do elementów kolekcji.
- Na listach jednokierunkowych można iterować tylko do przodu.
- Dołączanie i usuwanie elementów na końcu i na początku listy jest bardzo szybkie, ale dostanie się do elementu ze środka zabiera dużo czasu.



Kontenery sekwencyjne - listy jednokierunkowe

- Przykład:

```
forward_list<long> coll = { 2, 3, 5, 7, 11, 13};
…
coll.resize(9);
for (auto elem : coll) {
    cout << elem << ' ';
}
cout << endl;
```

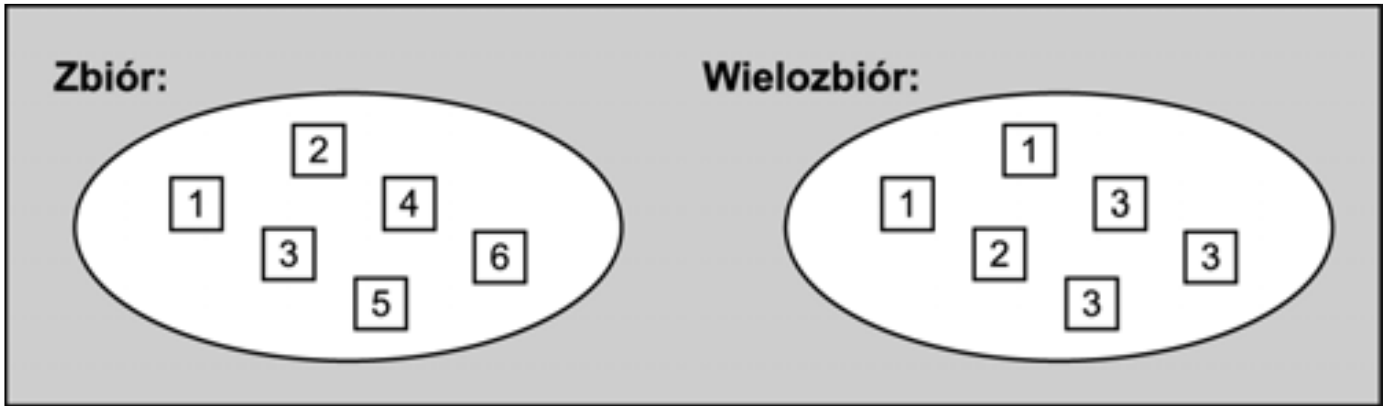
Kontenery sekwencyjne – łańcuchy i tablice

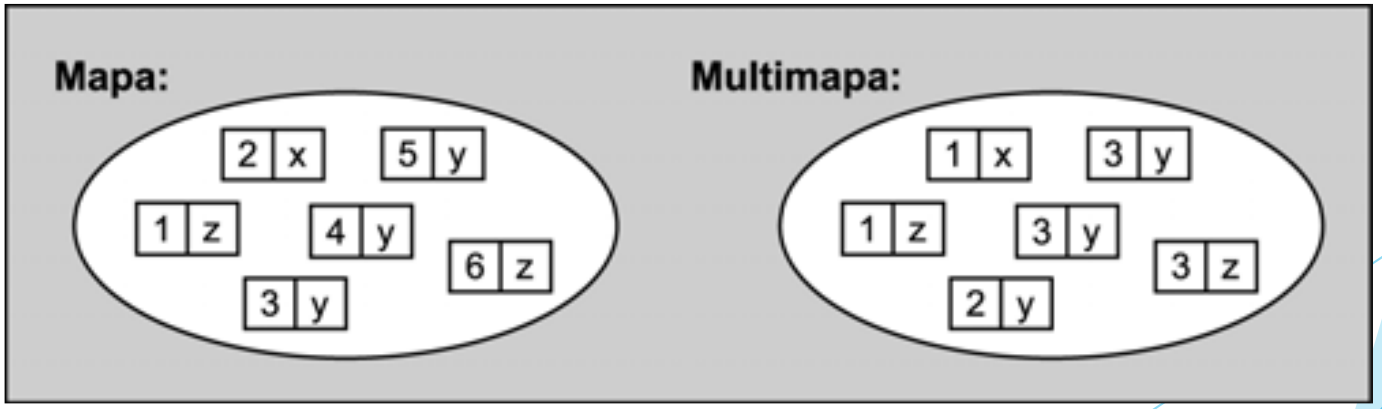
- Obiektów klas łańcuchowych, czyli `basic_string<>`, `string` i `wstring`, można używać jak kontenerów sekwencyjnych. Są one podobne w zachowaniu do wektorów.
- Innym rodzajem kontenera może być tablica. Nie jest to klasa i nie ma żadnych metod ale konstrukcja STL umożliwia uruchamianie na tablicach różnych algorytmów (tak jak na kontenerach).

Kontenery uporządkowane

- Kontenery uporządkowane wykonują automatycznie sortowanie swoich elementów.
- Asocjacyjne kontenery uporządkowane przechowują pary klucz-wartość (odpowiednio `first` i `second`) i sortowanie następuje po kluczach.
- Domyślnie elementy lub klucze są porządkowane przy pomocy operatora `<`.
- Kontenery uporządkowane są implementowane w postaci zrównoważonych drzew BST (drzewa czerwono-czarne).
- Wszystkie kontenery uporządkowane posiadają domyślny parametr wzorca służący sortowaniu (domyślnym jest operator `<`).
- Rodzaje kontenerów: zbiory `set<>`, wielozbiory `multiset<>`, mapy `map<>` i multimapy `multimap<>`.

Kontenery uporządkowane





Kontenery uporządkowane

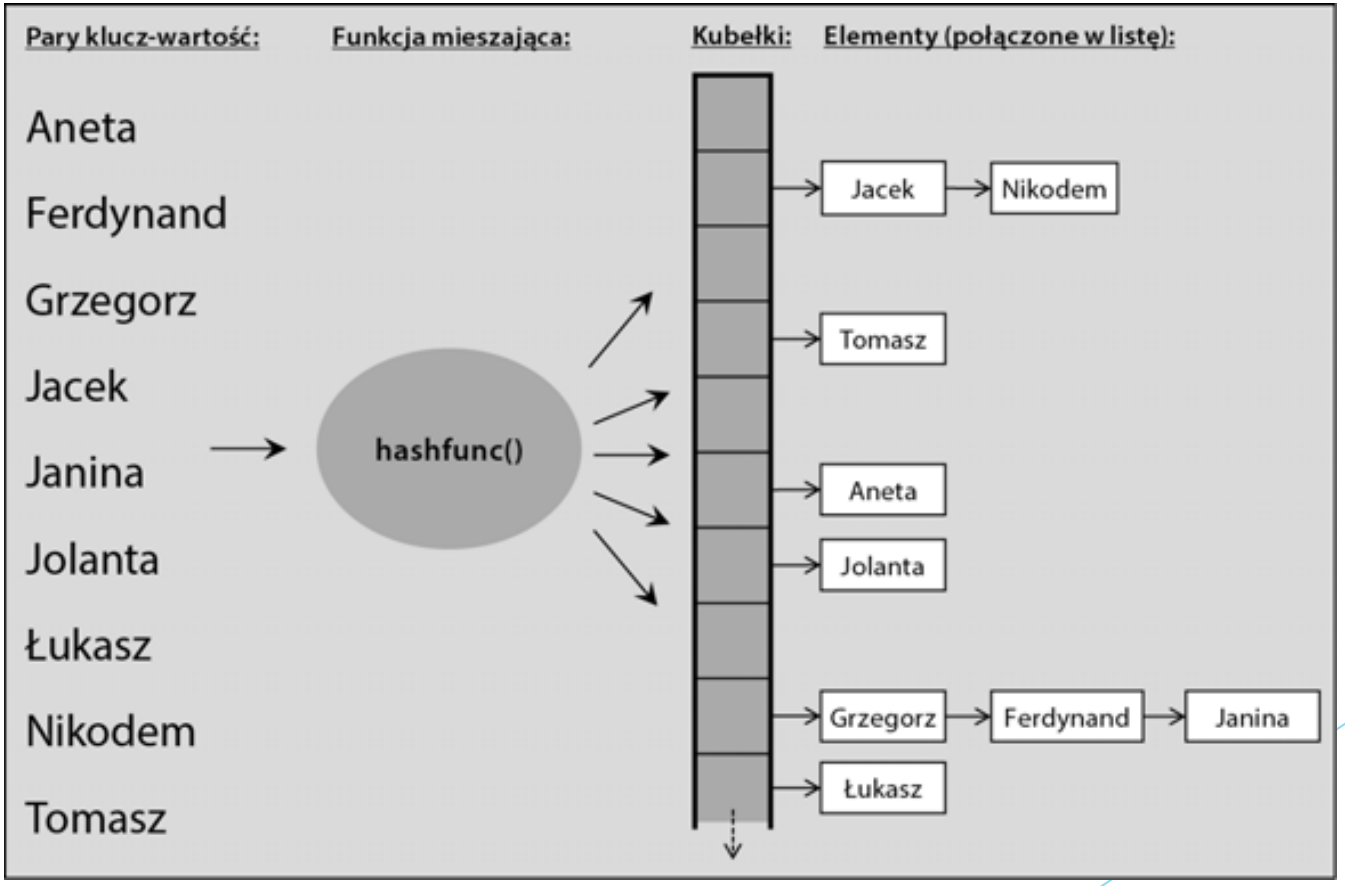
- Przykład:

```
multiset<string, greater<string>> cities {
    "Hanover", "Chicago", "Frankfurt",
    "Nowy Jork", "Toronto", "Londyn",
    "Frankfurt"
};
…
cities.insert({"Los Angeles",
    "Monachium", "Hanover", "Londyn"});
…
for (const auto& elem : cities)
    cout << elem << " ";
cout << endl;
```

Kontenery nieuporządkowane

- Kontenery nieuporządkowane przechowują elementy w sposób nieuporządkowany (brak kryterium sortowania).
- Asocjacyjne kontenery nieuporządkowane przechowują pary klucz-wartość (odpowiednio `first` i `second`).
- Kontenery nieuporządkowane są implementowane w postaci tablic z haszowaniem.
- Wszystkie kontenery uporządkowane posiadają domyślny parametr wzorca służący sortowaniu (domyślnym jest operator <).
- Rodzaje kontenerów: zbiory `unordered_set<>`, wielozbiory `unordered_multiset<>`, mapy `unordered_map<>` i multimapy `unordered_multimap<>`.

Kontenery nieuporządkowane



Kontenery nieuporządkowane

- Przykład:

```
unordered_map<string, double> coll {
    { "lolek", 9.9 },
    { "bolek", 11.77 }
};
…
// oblicz kwadraty wszystkich wartości
for (pair<const string,double>& elem : coll)
    elem.second *= elem.second;
```

Adaptatory kontenerów

- Adaptatory kontenerów to kontenery wykorzystujące ogólną strukturę innych kontenerów do realizacji pewnych specyficznych potrzeb.
- Adaptatorami kontenerów są stosy `stack<>`, kolejki `queue<>` i kolejki priorytetowe `priority_queue<>`.
- Przy definiowaniu takich kontenerów można podać jako drugi parametr typ kontenera do realizacji struktury (domyślnie jest to `vector<>`), na przykład:

```
stack<int, vector<int>> st;
queue<double> qu;
priority_queue<string, deque<string>, less<string>> pq;
```

Iteratory

- Iterator to specjalny obiekt, który potrafi iterować po elementach kolekcji.
- Iterator ma zaimplementowaną semantykę wskaźnika – posiada operator wyłuskania elementu *, operatory przechodzenia do elementów sąsiednich ++ i -- oraz operatory porównywania pozycji == i !=.

Iteratory

- Wszystkie kontenery udostępniają funkcje tworzące iteratory do nawigowania po ich elementach – funkcja `begin()` zwraca iterator wskazujący na pozycję z pierwszym elementem w kolekcji a funkcja `end()` zwraca iterator wskazujący pozycję za ostatnim elementem.
- Każdy kontener definiuje dwa typy iteratorów – *`kontener`*`::iterator` przeznaczony do iterowania po elementach z możliwością odczytu i zapisu oraz *`kontener`*`::const_iterator` przeznaczony do iterowania po elementach tylko z możliwością odczytu.

Iteratory

- Przykład 1:

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

- Przykład 2:

```
list<char> coll;
…
list<char>::iterator pos;
for (pos=coll.begin(); pos!=coll.end(); ++pos)
  *pos = toupper(*pos);
```