Kontenery

dr inż. Ireneusz Szcześniak

jesień 2017 roku

Kontenery

- Kontener w C++ to *generyczna* struktura danych.
- Przechowuje elementy jednego dowolnego typu.
- Nie pozwala na mieszanie przechowywanych typów.
- Kontenery przechowują elementy przez wartość.
- Części komplementarne: *iterator*, *algorytm*.
- Kontenery są implementowane przez klasy szablonowe.

Historia

- Kiedyś (początek lat 90.):
  - nowość, kierunek badań naukowych,
  - implementacja jako Standard Template Library (STL).
- Teraz:
  - niezbędne narzędzie,
  - w bibliotece standardowej od C++98.
- Nie ma wymówki: trzeba używać.
- Ale szasami trzeba użyć czegoś innego w starym (legacy) kodzie.

Typy kontenerów

- `std::vector<T>` - wektor: ciągłość w pamięci
- `std::list<T>` - lista, bez operatora swobodnego dostępu
- `std::deque<T>` - tablica, bez ciągłości w pamięci
- `std::map<K, V>` - tablica asocjacyjna (inaczej słownik)
- `std::multimap<K, V>` - słownik, klucze zduplikowane
- `std::set<T>` - zbiór
- `std::multiset<T>` - zbiór, klucze zduplikowane
- `std::priority_queue<T>` - kolejka priorytetowa

Kontenery mogą być zagnieżdżane: T może być kontenerem.

Typy kontenerów - wady i zalety

- `std::vector<T>` - najlepszy, jeżeli potrzebujemy swobodnego dostępu (`operator []`), ale rzadko zmieniamy rozmiar wektora, wstawiamy albo usuwamy elementy. Jest to po prostu tablica, którą możemy swobodnie i wygodnie zmieniać.
- `std::list<T>` - najlepsza, jeżeli często zmieniamy rozmiar, często dodajemy albo usuwamy elementy, wystarczy dostęp iteracyjny i nie jest nam potrzebny swobodny dostęp (nie ma `operator []`).
- `std::deque<T>` - najlepszy, jeżeli często zmianiamy rozmiar, często dodajemy albo usuwamy elementy, ale ciągle potrzebujemy szybkiego swobodnego dostępu (`operator []` jest).

Iteratory

- Pozwalają na dostęp do elementów kontenera.
- Implementacja: wskaźniki obudowane w klasy.
- Do funkcji przekazujemy przez wartość, nie referencję.
- Dla kontenera T, iterator to: `T::iterator`.
- Iterator "const" obiektów stałych: `T::const_iterator`.
- Używaj iteratora "const", jeżeli nie modyfikujesz kontenera.
- Podstawowe operacje: `++i`, `i++`.
- Podstawowe funkcje: `T::begin()`, `T::end()`.
- Różne funkcje (`find`, `insert`) zwracają iteratory.

Iteratory odwrotne

- Pozwalają na iterację od końca po elementach kontenera.
- Typ: `T::reverse_iterator` i `T::const_reverse_iterator`
- Skomplikowane, bardzo trudne w użyciu i...
- dlatego nie mają większego znaczenia praktycznego.
- Przestroga: lepiej nie używać!

Algorytmy

- Algorytmy dla różnych kontenerów:
  - sortowanie elementów (lepszej implementacji nie znajdziecie),
  - wyszukiwanie elementów,
  - iterowanie po elementach,
  - usuwanie i dodawanie elementów.
- Działają dla różnych kontenerów, bo kontenery mają ten sam interfejs:
  - nazwy typów (np. `T::data_value`),
  - nazwy funkcji (np. `T::begin()`).

std::pair<A, B>

- `#include <utility>`
- Para p obiektów typów A i B: `std::pair<A, B> p;`
- p.first - pierwszy element, p.second - drugi element
- Globalna funkcja szablonowa `std::make_pair` pozwala na tworzenie pary bez podania typów, kompilator wywnioskuje typy sam:
  `std::make_pair(1, "test");`
- Funkcja `tie` pozwala na wygodne przypisanie wartości z pary do osobnych zmiennych:
  `std::tie(f, s) = p;`
- Przeniesienie pary polega na przeniesieniu każdego z obiektów składowych.
- Ma zdefiniowane globalne operatory: `!= < == > <= >=`

Obiekt funkcyjny

- Obiekt funkcyjny nazywany jest też funktorem.
- Obiekt klasy, w której zdefiniowany jest operator wywołania funkcji: `operator()`
- Ten operator musi być `non-static` i najlepiej `const`.
- Ten operator można przeciążyć na wiele sposobów, ale nas interesuje operator, który można wykorzystać do porównania obiektów pewnej klasy A, elementów kontenera:
  `bool operator () (const A &a1, const A &a2) const;`
- Kompilator będzie się starał wkompilować inline tę funkję.
- ZALETA: Obiektowi funkcyjnemu możemy podać dodatkowe dane w konstruktorze, potem używane w operatorze wywołania funkcji.

Obiekt funkcyjny - przykład

```
struct CMP
{
  bool m_order;
  CMP(bool order): m_order(order) {}
  bool operator () (const A &e1,
                    const A &e2) const
  {
    bool s = e1.m_text < e2.m_text;
    return m_order ? !s : s;
  }
};
```

Pętle iteracyjne po staremu

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> a;
  a.push_back(1);
  a.push_back(2);
  a.push_back(3);
  for(std::vector<int>::iterator i = a.begin();
      i != a.end(); ++i)
    std::cout << *i << std::endl;
}
```

Pętle iteracyjne po nowemu

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> a = {1, 2, 3};
  for(const auto &e: a)
    std::cout << e << std::endl;
}
```

Podsumowanie

- Unikać implementacji własnych struktur danych!
- Używać biblioteki standardowej!
- Kontenery biblioteki standardowej pozwalają na tworzenie dowolnych struktur danych.
- Algorytmy standardowe lepsze od własnych tandetnych implementacji.

Dziękuję za uwagę.