STL Iteratory i Kontenery

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
AGH University of Science and Technology

8 listopada 2018

Kontenery

Czym jest kontener

- To klasa, której przeznaczeniem jest przechowywanie wartości w sposób zorganizowany (struktura danych)
- Ma zdefiniowane operacje do właściwego przetwarzania danych – zgodnego z założeniami
- Najprostszy kontener - tablica

Rodzaje kontenerów

- Sekwencyjne – przechowujące elementy w kolejności, w jakiej zostały napisane np. `vector, deque, list`
- Asocjacyjne – nie zachowują kolejności dodawania elementów, tylko sortują je według określonego kryterium np. `map, set`

Rodzaje kontenerów

- Uporządkowane – używają porządku obiektów np. `map, set`
- Nieuporządkowane – używają funkcji skrótu np. `unordered_map, unordered_set`

przykład 1

Przykłady kontenerów

vector

- Reprezentuje szereg elementów określonego typu
- Dostęp do elementów przez indeksy
- Można określić początkowy rozmiar, który można potem zmienić
- Dodawanie elementów przez metodę `push_back (emplace_back)`

vector

- Nie gwarantuje sprawdzania zakresu
- Używamy go zamiast zwykłej tablicy, oraz gdy chcemy sprawnie odwoływać się do elementów lub dodawać je na koniec – dodawanie w środku/usuwanie jest wolniejsze w porównaniu do innych kontenerów

list

- Lista dwukierunkowa, `forward_list` – lista jednokierunkowa
- Używa jej się do przechowywania danych, które mają być często dodawane/usuwane – kontener wykonuje te operacje bez przesuwania pozostałych elementów
- Nie można używać operatora[] – aby odczytać konkretny element trzeba do niego przeiterować

list

- Lista jest odpowiednia dla danych, które mają być często przeszukiwane (oraz iterowane)
- Posiada własne funkcje sortujące i wyszukujące – należy używać ich, zamiast pochodzących z biblioteki z algorytmami ze względu na organizacje pamięci listy
- Dla małych ilości elementów bardziej wydajne jest używanie wektora

deque

- Kolejka dwustronna
- Udostępnia dostęp do elementów za pomocą operatora []
- Przechowywana w pamięci w sposób nieciągły – usunięcie elementu nie wymaga kopiowania jak w przypadku wektora
- Wybieramy ten kontener, gdy potrzebujemy możliwości szybkiego dodawania/usuwania elementów z początku/końca kontenera

`set`

- Zbiór, przechowuje po jednym elemencie danego typu
- Istnieje również `multiset`, który przechowuje obiekty tyle razy, ile je się dodało
- Używając przeszukiwania binarnego kontener jest w stanie szybko stwierdzić, czy znajduje się w nim wybrany element

map

- Inne nazwy: słownik, tablica asocjacyjna
- Jest kontenerem par zoptymalizowanych pod względem wyszukiwania, czyli klucza, który jest indeksem oraz wartości
- Udostępnia dostęp operatorem[ ] oraz metodą `at()`, która sprawdza czy dany klucz istnieje
- Dereferencja zwraca parę – musimy dodatkowo wskazać interesujący nas parametr

przykład 2

unordered_map, unordered_set

- C++11
- Kontenery mają takie same zachowanie, jak ich uporządkowane odpowiedniki
- Nie korzystają z sortowania kluczy/wartości
- Używają funkcji haszującej, dzięki którym operacje wstawiania i wyszukiwania wykonują się w czasie stałym

Funkcja haszująca

- Jest to funkcja przypisująca każdemu elementowi tzw skrót nieodwracalny, czyli quasi-losową wartość, o stałej długości
- Aby odnaleźć interesujący nas element, obliczamy wartość funkcji, a następnie sięgamy do odpowiedniego miejsca w pamięci

Adaptery kontenerów

- Zapewniają nowy interfejs kontenerom, który umożliwia im inny zakres operacji
- np. `stack` (struktura LIFO), `queue` (struktura FIFO) – oparte na `deque`

Definiowanie kontenerów

Definiowanie własnych kontenerów

- Aby stworzyć własny kontener, wzorując się na kontenerach STL, musimy utworzyć szablon klasy
- Powinien on mieć zdefiniowane wybrane przez nas operacje, które będziemy chcieli wykonywać na kontenerze

Definiowanie własnych kontenerów - wzór

```
template
  <class T,
  class std::allocator<T> >
 class NewContainer {};
```

» `T` będzie typem przechowywanym w kontenerze, a `std::allocator<T>` alokatorem, czyli klasą która służy do przydzielania pamięci dla kontenerów

przykład 3

Definiowanie własnych adapterów

- Adapter również jest szablonem
- Jako parametr podajemy kontener, który chcemy adaptować, następnie dokonujemy restrykcji/modyfikacji zgodnie z naszym zamysłem

Definiowanie własnych adapterów - wzór

```
» template<
        class T,
        class Container = std::deque<T>
  >
     class New Adapter;
```

przykład 4

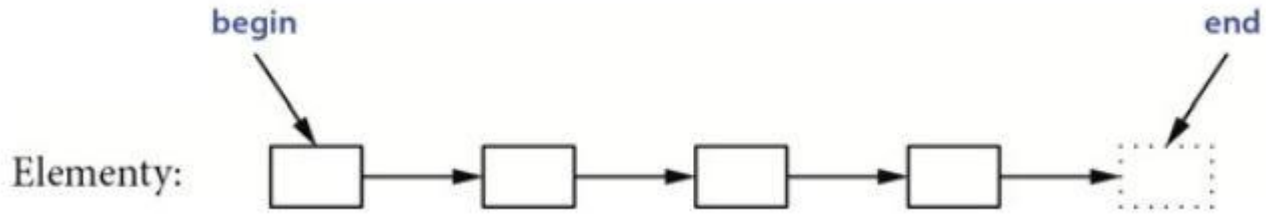
Iteratory

Czym jest iterator?

- Iterator jest obiektem, który umożliwia poruszanie się po elementach kontenerów i odczytywanie ich zawartości
- Można go inkrementować
- Minimalizuje zależności algorytmów od struktur danych, na których działa

Czym jest iterator?

» Każdy kontener udostępnia dwie metody `begin()` i `end()`, zwracające iterator na początek i koniec kontenera.



przykład 5

Kategorie iteratorów

- Iterator wejściowy
- Iterator wyjściowy
- Iterator jednokierunkowy
- Iterator dwukierunkowy
- Iterator o dostępie swobodnym

Iterator wejściowy

- Umożliwia przeglądanie do przodu (operator++) oraz odczyt elementów
- Można je porównywać za pomocą operatorów == oraz !=
- Dostępny w strumieniu `istream`

Iterator wyjściowy

- Umożliwia przeglądanie sekwencji do przodu (operator++) oraz zapis elementów – odczyt jest niemożliwy
- Można je porównywać za pomocą operatorów == oraz !=
- Dostępny w strumieniu `ostream`

Iterator jednokierunkowy

- Umożliwia przeglądanie do przodu oraz odczyt i zapis elementów, jeśli nie są `const`
- Można je porównywać za pomocą operatorów == oraz !=
- W odróżnieniu do iteratorów wejściowych/wyjściowych mogą przetwarzać dany element wielokrotnie

Iterator dwukierunkowy

- Umożliwia przeglądanie do przodu (operator++) i do tyłu (operator--) oraz odczyt i zapis elementów, jeśli nie są `const`
- Pozostałe cechy takie same jak u iteratora jednokierunkowego
- Używany np. w liście

Iterator o dostępie swobodnym

- Umożliwia przeglądanie do przodu i do tyłu oraz odczyt i zapis elementów, jeśli nie są `const`
- Można go indeksować za pomocą operatora []
- Udostępnia działania artymetyki iteratorów
- Używany np. w wektorze

Arytmetyka iteratorów

- Zwiększanie/zmniejszanie o liczbę całkowitą operatorem +/-, nie tylko ++/--
- Obliczanie różnicy między dwoma iteratorami odejmując je
- Porównywanie operatorami ==, !=, <, <=, >, >=

Podsumowanie

Iterator category				Defined operations
RandomAccessIterator	*BidirectionalIterator*	*ForwardIterator*	*InputIterator*	• read • increment (without multiple passes)
				• increment (with multiple passes)
				• decrement
				• random access
Iterators that fall into one of the above categories and also meet the requirements of *OutputIterator* are called mutable iterators.				
OutputIterator				• write • increment (without multiple passes)

Źródło: https://italiancpp.github.io/meetup-may2017.html

Cechy iteratorów

Cechy iteratorów

- Różne kategorie iteratorów mają różne możliwości, aby je w pełni wykorzystać powinniśmy dostosowywać nasz działania do obecnie używanego rodzaju
- Pomagają nam w tym `iterator_traits` oraz `iterator_tag`, czyli mechanizm, umożliwiający sprawdzenie, który rodzaj iteratora jest używany

iterator_traits

- Są to cechy definiujące własności iteratorów takie jak: typ, który może być wskazywany, typ wskaźnika, typ referencji, typ różnicy iteratorów i kategoria iteratora
- Kategorie iteratora są zdefiniowane w bibliotece standardowej jako 5 pustych klas
- Tworząc własny iterator należy zdefiniować wszystkie cechy

przykład 6

iterator_tag

```
namespace std {
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag: public
     input_iterator_tag {};
  struct bidirectional_iterator_tag:
     public forward_iterator_tag {};
  struct random_access_iterator_tag:
     public bidirectional_iterator_tag {};
  }
```

Funkcje pomocnicze i ich implementacja

Funkcje pomocnicze

- `void advance (pos, n)`
- `long distance (first, second)`
- `void iter_swap (first, second)`
- `next (pos, n)`
- `prev (pos, n)`

Funkcja `advance`

- Przesuwa iterator o `n` pozycji do przodu
- Funkcja nie sprawdza, czy przekroczono zakres
- Dla iteratorów dwukierunkowych i swobodnego dostępu `n` może być <0
- Dzięki `iterator traits` funkcja sama wybiera najlepszy sposób wykonania przesunięcia

Funkcja `distance`

- Zwraca odległość między iteratorami `first` i `second`
- Zwykle typu `int`
- Jeśli `second` nie jest osiągalny z `first` zachowanie jest niezdefiniowane
- Funkcja sama wybiera najlepszy sposób do wyznaczenia odległości

Funkcja `iter_swap`

- Zamienia wartość elementów, na które wskazują iteratory `first` i `second`

Funkcja `next`

- Zwraca iterator wskazujący `n` pozycji w przód
- Funkcja nie zmienia argumentu, tylko zwraca przesuniętą kopię
- Nie sprawdza czy przekroczono zakres
- Dostępna od C++11

Funkcja `prev`

- Zwraca iterator wskazujący `n` pozycji w tył
- Nie sprawdza czy przekroczono zakres
- Dostępna od C++11

przykład 7

Adaptatory iteratorów

Adaptory Iteratorów

- Adaptatorami nazywamy specjalne iteratory umożliwiające działanie w specjalnych trybach
- Znajdują się w nagłówku `<iterator>`

Iteracja wstecz

- `reverse_iterator`
- Iteratorem odwrotnym nazywamy iterator, który zmienia kierunek iterowania na przeciwny
- Metody `rbegin()` i `rend()`

Wstawianie na końcu

- `back_insert_iterator`
- Wstawia wartość do sekwencji, zwiększając rozmiar kontenera
- Iterator chroni przed nadpisaniem elementów
- Do wstawienia używa funkcji `push_back()`

Wstawianie na początku

- `front_insert_iterator`
- Podobnie jak `back_insert_iterator` chroni pamięć
- Wstawia element na początku, za pomocą funkcji `push_front()`

Wstawianie w dowolnym miejscu

- `insert_iterator`
- Wstawia element przed elementem wskazywanym
- Wymaga przekazania iteratora dwukierunkowego, wskazującego element w kontenerze

Przenoszenie zamiast kopiowania

- `move_iterator`
- Przenosi element przy odczycie zamiast go kopiować
- Udostępnia ten sam zestaw operacji, co iterator z którego został utworzony

Zapis w niezainicjowanej pamięci

- `raw_storage_iterator`
- W standardowych algorytmach przyjmuje się, że elementy są zainicjowane, więc do zapisu używa się przypisywania – nie można użyć bezpośrednio niezainicjowanej pamięci
- Iterator `raw_storage_iterator` inicjuje pamięć, zamiast przypisywać
- Operacja mniej kosztowna

przykład 8

Definiowanie iteratorów

Definiowanie iteratorów

- W kontenerze powinien być zdefiniowany iterator zwykły oraz w wersji `const` odpowiedniej, dopasowanej kategorii
- Należy pamiętać o zdefiniowaniu iterator traits oraz odpowiednich metod, które dostarczą działań zgodnych z kategorią

przykład 9

Źródła

- The C++ Programming Language, Bjarne Stroustrup
- https://en.cppreference.com/
- https://italiancpp.github.io/meetup-may2017.html
- https://pl.wikipedia.org/
- https://pl.wikibooks.org/
- http://sun.aei.polsl.pl/~sdeor//students/stl/stl_w05.pdf
- https://golinski.faculty.wmi.amu.edu.pl/ppr/11.html
- https://cpphelp.com/slist/presentation.ppt

github.com/karmazynow-a/adv2018

zadania, prezentacja, przykłady