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 obiekt, który przechowuje inne obiekty 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`

- 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
- Jest dynamiczny - można określić początkowy rozmiar, który można potem zmienić
- Dodawanie elementów przez metodę `push_back` `(emplace_back)`
- Nie gwarantuje sprawdzania zakresu

vector

- Wartości są przechowywane w sposób ciągły – usuwając element ze środka musimy skopiować resztę elementów, aby zachować ciągłość
- 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
- Elementy są przechowywane w pamięci w sposób nieciągły
- 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

forward_list

- C++11
- Mimo że jest „rodzajem” listy, to nie jest ona adaptatorem
- Przechowuje elementy w sposób nieciągły
- Pozwala na wstawianie i usuwanie w dowolnym miejscu
- Bezpośredni dostęp (operator[ ]) nie jest dostępny
- Zużywa mniej pamięci niż `list` kosztem możliwości iteracji w obu kierunkach (lista single-linked z C)
- Aby dostać się do n-tego elementu, należy przeiterować od pierwszego
- Brak metody `size` !!

deque

- Kolejka dwustronna
- Można dodawać elementy w obu kierunkach – dodawanie elementów na początku/końcu jest bardzo szybkie
- 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

array

- C++11
- Stworzona na wzór tablicy z C
- Jest to tablica statyczna – ilość elementów jest stała, ustalona w czasie kompilacji, ale możemy zmieniać wartości elementów
- Udostępnia swobodny dostęp do elementów (operator[])
- Liczba elementów jest elementem składowym: `array<char,2>` i `array<char,4>` to dwa różne typy – niemożliwe jest przypisanie czy porównanie

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
- Istnieje również `unordered_multimap` i `unordered_multiset`

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 domyślnie na `deque`, lub `priority_queue` (kolejka priorytetowa) – oparta na `vector`
- Możemy ich używać nie bazując na domyślnym kontenerze, którego adaptują, lecz wpisać wybrany kontener, jako parametr

stack

- Stos, LIFO
- Dodajemy tylko na szczyt stosu, i tylko stamtąd zdejmujemy
- Kontener na którym bazuje musi być sekwencyjny i posiadać metody: `back() push_back(), pop_back()`
- Domyślnie jest `deque`, ale `vector` i `list` też spełniają te wymagania

queue

- Kolejka, FIFO
- Dodajemy tylko na początek, i usuwamy tylko z końca
- Kontener na którym bazuje musi być sekwencyjny i posiadać metody: `back()`, `front()`, `push_back()` i `pop_front()`
- Te wymagania spełniają `deque` i `list`

priority_queue

- Kolejka priorytetowa
- Wkładamy elementy do kolejki, a zdejmujemy pierwszy, największy
- Kontener na którym bazuje musi być sekwencyjny z iteratorem random access i posiadać metody: `front()`, `push_back()` i `pop_front()`
- Te wymagania spełniają `deque` i `vector`
- Dodatkowo, szablon STL przyjmuje jako parametr komparator, wg którego ustawiane są elementy

Definiowanie kontenerów

Definiowanie własnych kontenerów

- Aby stworzyć własny kontener, wzorując się na kontenerach STL, musimy utworzyć szablon klasy

```
template
        <class T,
        class A = 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
- Kontenery asocjacyjne mają dodatkowe parametry, takie jak comparator, hash, klucz etc

Definiowanie własnych kontenerów - standard

- Tworząc własny kontener, powinniśmy spełnić wymagania zapisane w standardzie:
  - stosowanie alokatora,
  - zdefiniowanie typów kontenera,
  - zaimplementowanie funkcji typowych dla wybranej struktury danych ,
  - zdefiniowanie iteratora (standardowego i const)

Definiowanie własnych kontenerów - typedef

- Jeśli chcemy aby nasz kontener spełniał standard STL musimy zdefiniować typy kontenera, między innymi:

```
typedef A allocator_type;
typedef typename A::value_type value_type;
typedef typename A::reference reference;
typedef typename A::const_reference
const_reference;
typedef typename A::difference_type
difference_type;
typedef typename A::size_type size_type;
typedef Compare value_compare;
```

Definiowanie własnych kontenerów - alokatory

- Domyśnie używany alokator w kontenerach to :

```
template< class T >
struct allocator;
```

- Używa on operatorów `new` i `delete` do tworzenia nowych obiektów, aczkolwiek standard nie mówi jak często i kiedy mają być one wywoływane
- Udostępnia metody `allocate` i `deallocate` – rezerwujące pamięć oraz `construct` i `destroy` – tworzące/niszczące obiekt w zaalokowanej pamięci
- Standard wymaga aby obiekty, które przechowuje kontener były tworzone i niszczone za pomocą alokatora

Definiowanie własnych kontenerów - komparatory

- Jeśli używamy kontenerów asocjacyjnych, w szablonie powinien się znaleźć również komparator, który będzie ustalał kolejność elementów w kontenerze
- Domyślnym komparatorem jest `std::less<T>`, ale można też użyć własnego, zdefiniowanego funktora, który porównuje wartości

```
bool operator()( const T& lhs, const T& rhs )
const;
```

- Kontenery nieuporządkowane nie mają komparatorów, ale parametry potrzebne do hashowania

Definiowanie własnych kontenerów - wymogi

- Istnieją specjalne wymogi, które mają spełniać szablony klas, aby można je było nazwać kontenerami
- W standardach języka znajdują się rozdziały *Container Requirements*
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf - rodział 23

- Każdy kontener powinien również zawierać iteratory. Czym one są i jak je zdefiniować w kontenerze dowiemy się w drugiej części prezentacji

Definiowanie własnych adapterów

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

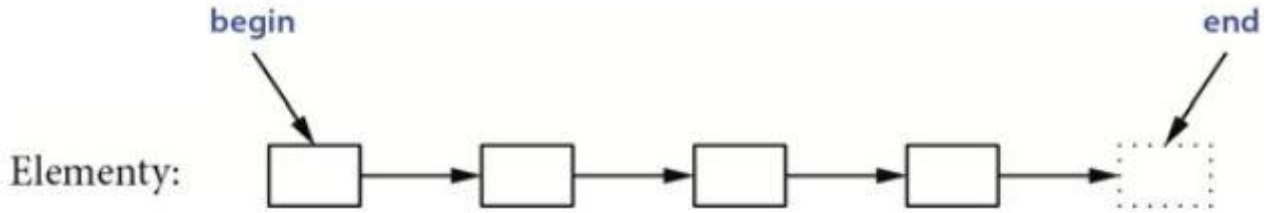
```
template<
        class T,
        class Container = std::deque<T> >
class New Adapter;
```

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
- 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
- Używany np. w `forward_list`

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 `list`

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 `vector, deque` lub `array`

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://en.cppreference.com/w/cpp/iterator

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

- `template< class InputIt, class Distance > void advance( InputIt& it, Distance n );`
- `template< class InputIt > typename std::iterator_traits<InputIt>::difference_type distance( InputIt first, InputIt last );`
- `template< class ForwardIt1, class ForwardIt2 > void iter_swap( ForwardIt1 a, ForwardIt2 b );`
- `template< class ForwardIt > ForwardIt next( ForwardIt it, typename std::iterator_traits<ForwardIt>::difference_type n = 1 );`

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
- Istnieje również funkcja `prev` działająca analogicznie, lecz zwracająca iterator przesunięty o `n` pozycji w tył

przykład 7

Adaptatory iteratorów

Adaptory Iteratorów

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

```
reverse_iterator,
insert_iterator,
back_insert_iterator,
move_iterator,
raw_storage_iterator
```

Iteracja wstecz

- `reverse_iterator`
- Iteratorem odwrotnym nazywamy iterator, który zmienia kierunek iterowania na przeciwny
- Udostępnia metody `rbegin()`, która zwaraca iterator wskazujący na ostatni element kontenera i `rend()` - przed pierwszym
- Powinien być `bidirectional` albo `random_access`
- Posiada metodę `base`, która zwraca kryjący się iterator – jeśli zmniejszymy iterator `base` o jeden otrzymamy tą samą wartość która jest pod iteratorem

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
- Używa się go zazwyczaj w algorytmach, takich jak `accumulate, copy`

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

» Najprostrzym sposobem na zdefiniowanie iteratora jest stworzenie szablonu dziedziczącego po podstawowym iteratorze:

```
template <class Category, class T, class Distance =
ptrdiff_t, class Pointer = T*, class Reference = T&>
struct iterator{
    typedef T value_type;
    typedef Distance difference_type;
    typedef Pointer pointer;
    typedef Reference reference;
    typedef Category iterator_category };
```

przykład 9

Definiowanie iteratorów przez dziedziczenie

- W takim przypadku szablon naszego iteratora będzie wyglądał np:

```
template< class ValueType, class NodeType >
class my_iterator : std::iterator<
std::bidirectional_iterator_tag, T >{};
```

- Nowy szablon dziedziczy po specjalizacji podstwowego iteratora
- Inną opcją jest stworzenie szablonu na wzór podstawowego i dostarczenie implementacji dla wszystkich 5 kategorii (jako specjalizacje szablonu)

przykład 10

Definiowanie iteratorów w kontenerze

- W kontenerze powinien być zdefiniowany iterator zwykły oraz w wersji `const` odpowiedniej, dopasowanej kategorii
- Należy pamiętać o `iterator_traits` oraz zdefiniowaniu w iteratorze odpowiednich metod, które dostarczą działań zgodnych z kategorią
- Wymagana jest również implementacja funkcji `begin()` i `end()` (oraz ich wersji `const` i `reverse`) w ciele klasy

przykład 9

Źródła

- The C++ Programming Language, Bjarne Stroustrup
- Programming languages – C++ International Standard
- 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