Paradygmaty programowania obiektowego

Iteratory w programowaniu obiektowym

dr inż. Radosław Idzikowski

Co to jest iterator?

Iterator

Obiekt służący do iterowania po elementach sekwencji lub kontenera. W języku C++ iterator jest szczególnym rodzajem wskaźnika, który umożliwia iterowanie po kontenerach `STL` (*Standard Template Library*)

Kontener

Struktura danych, która przechowuje i zarządza kolekcją elementów o określonym typie. Kontenery zapewniają dynamiczne zarządzanie pamięcią oraz różne algorytmy dostępu do przechowywanych danych.

Rodzaje kontenerów w C++

Wektor (`vector`)

Dynamiczna tablica, która przechowuje elementy w jednym ciągłym obszarze pamięci i umożliwia dostęp do nich za pomocą operatora nawiasów kwadratowych ([]). Wektor umożliwia szybki dostęp do elementów za pomocą iteratorów. Ponadto rozmiar wektora może się dynamicznie zmieniać.

- `push_back(elem)` - dodaj element na koniec,
- `pop_back()` - usuń ostatni element wektora,
- `size()` - zwróć aktualny rozmiar wektora,
- `empty()` - sprawdź, czy wektor jest pusty,
- `clear()` - usuń wszystkie elementy z wektora,
- `resize(new_size)` - zmień rozmiar wektora na nowy rozmiar,
- `reserve(new_capacity)` - zarezerwuj pamięć na co najmniej określoną liczbę elementów,
- `begin()` - zwróć iterator do pierwszego elementu wektora,
- `end()` - zwróć iterator do elementu za ostatnim elementem wektora.

Rodzaje kontenerów w C++

Lista (`list`)

Podwójnie powiązana lista, która pozwala na dodawanie i usuwanie elementów w każdym miejscu listy. Listy umożliwiają szybki dostęp do elementów za pomocą iteratorów, ale nie umożliwiają szybkiego dostępu do elementów za pomocą operatora nawiasów kwadratowych.

- `push_front(elem)` - dodaj element na początek listy,
- `pop_front()` - usuń pierwszy element listy,
- `push_back(elem)` - dodaj element na koniec listy,
- `pop_back()` - usuń ostatni element listy,
- `size()` - zwróć aktualny rozmiar listy,
- `empty()` - sprawdź, czy lista jest pusta,
- `clear()` - usuń wszystkie elementy z listy,
- `begin()` - zwróć iterator do pierwszego elementu listy,
- `end()` - zwróć iterator do elementu za ostatnim elementem listy,
- `insert(iterator, elem)` - dodaj element w miejscu wskazywanym przez iterator,
- `erase(iterator)` - usuń element w miejscu wskazywanym przez iterator,
- `remove(value)` - usuń wszystkie elementy o wartości `value`.

Rodzaje kontenerów w C++

Kolejka (queue)

FIFO (*First In, First Out*), która umożliwia dodawanie elementów na koniec kolejki i usuwanie elementów z początku kolejki. Kolejki zaimplementowane są za pomocą list lub wektorów.

- `push(elem)` - dodaj element do kolejki,
- `pop()` - usuń pierwszy element kolejki,
- `front()` - zwróć pierwszy element kolejki,
- `back()` - zwróć ostatni element kolejki,
- `size()` - zwróć aktualny rozmiar kolejki,
- `empty()` - sprawdź, czy kolejka jest pusta.

Rodzaje kontenerów w C++

Stos (`stack`)

LIFO (*Last In, First Out*), która umożliwia dodawanie elementów na wierzch stosu i usuwanie elementów z wierzchu stosu. Stosy zaimplementowane są za pomocą list lub wektorów.

- `push(elem)` - dodaj element na wierzch stosu,
- `pop()` - usuń element z wierzchu stosu,
- `top()` - zwróć element z wierzchu stosu,
- `size()` - zwróć aktualny rozmiar stosu,
- `empty()` - sprawdź, czy stos jest pusty.

Rodzaje kontenerów w C++

Mapa (`map`)

Asocjacyjny kontener, który umożliwia przechowywanie par klucz-wartość. Mapa umożliwia szybkie wyszukiwanie wartości za pomocą klucza.

- `insert(pair)` - dodaj parę klucz-wartość do mapy,
- `erase(key)` - usuń element o podanym kluczu z mapy,
- `find(key)` - zwróć iterator do elementu o podanym kluczu,
- `operator[](key)` - dostęp do elementu o podanym kluczu,
- `size()` - zwróć aktualny rozmiar mapy,
- `empty()` - sprawdź, czy mapa jest pusta.

Rodzaje kontenerów w C++

Zbiór (`set`)

Zbiór przechowuje unikalne elementy w porządku rosnącym. Zbiory umożliwiają szybkie wyszukiwanie elementów.

- `insert(elem)` - dodaj element do zbioru,
- `erase(elem)` - usuń element z zbioru,
- `find(elem)` - zwróć iterator do elementu o podanej wartości,
- `size()` - zwróć aktualny rozmiar zbioru,
- `empty()` - sprawdź, czy zbiór jest pusty.

Rodzaje kontenerów w C++

Krotka (`tuple`)

Kontener, który umożliwia przechowywanie wielu wartości różnych typów w jednej strukturze. Krotki umożliwiają łatwe przechowywanie i manipulowanie zbiorem wartości o różnych typach.

- `std::tuple_element<i,T>::type` - zwraca typ i-tego elementu krotki T,
- `std::tuple_size<T>::value` - zwraca ilość elementów w krotce T,
- `std::make_tuple(args...)` - tworzy krotkę z wartościami args,
- `std::tuple_cat(t1, t2)` - scala dwie krotki t1 i t2 w jedną krotkę,
- `std::get<i>(t)` - zwraca wartość i-tego elementu krotki t.

Rodzaje iteratorów w C++

- `iterator` - podstawowy rodzaj iteratora umożliwiający poruszanie się po sekwencjach,
- `const_iterator` - iterator umożliwiający jedynie odczyt wartości,
- `reverse_iterator` - iterator poruszający się w odwrotnym kierunku po sekwencji,
- `const_reverse_iterator` - reverse iterator umożliwiający jedynie odczyt wartości,
- `move_iterator` - iterator umożliwiający przeniesienie zasobów z jednego kontenera do drugiego.

Przykład użycia iteratorów na wektorze

```cpp
#include <iostream>
#include <vector>
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    for (auto it = vec.begin(); it != vec.end(); it++) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    for (auto x : vec) {
        std::cout << x << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Przykład użycia iteratorów na liście

```
#include <iostream>
#include <list>
int main() {
    std::list<int> lst = {1, 2, 3, 4, 5};
    for (auto it = lst.begin(); it != lst.end(); it++) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    for (auto it = lst.rbegin(); it != lst.rend(); it++) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Przykład użycia iteratorów na mapie

```
#include <iostream>
#include <map>
int main() {
    std::map<std::string, int> m = {
        {"a", 1},
        {"b", 2},
        {"c", 3}
    };
    for (auto it = m.begin(); it != m.end(); it++) {
        std::cout << it->first << ": " << it->second << std::endl;
    }
    std::cout << std::endl;
    return 0;
}
```

Właściwości iteratorów

Iterator w C++ posiada wiele właściwości, które czynią go bardzo przydatnym narzędziem w programowaniu. Oto niektóre z tych właściwości:

- uniwersalność - iterator może działać na różnych rodzajach kontenerów i pozwala na ich dynamiczną modyfikację,
- przezroczystość - iterator jest zwykłym wskaźnikiem, co pozwala na używanie znanych operacji takich jak $*$ i ->,
- efektywność - iterator umożliwia szybkie przeszukiwanie sekwencji, bez konieczności tworzenia dodatkowych struktur danych,
- bezpieczeństwo - iterator chroni przed dostępem do niedozwolonych obszarów pamięci, co minimalizuje ryzyko błędów,
- elastyczność - różne rodzaje iteratorów pozwalają na specyficzne operacje i dostosowanie do potrzeb programisty.

Iterator a wskaźnik

- typ - iterator to typ obiektowy, podczas gdy wskaźnik to typ podstawowy.
- przestrzeń nazw - iteratory znajdują się w przestrzeni nazw standardowej std, podczas gdy wskaźniki nie są związane z żadną przestrzenią nazw.
- operacje - iteratory umożliwiają różne operacje, takie jak przemieszczanie się po elementach kontenera, porównywanie dwóch iteratorów itp. Wskaźniki pozwalają na wykonywanie podobnych operacji, ale nie są tak rozbudowane jak iteratory.
- bezpieczeństwo typów - iteratory są bezpieczniejsze typowo niż wskaźniki. Na przykład, iterator zdefiniowany dla listy nie może zostać użyty z wektorem. Wskaźnik nie ma takich ograniczeń.
- wyjątki - iteratory zapewniają wyjątki, gdy dojdzie do błędu, takiego jak próba odczytu elementu poza zakresem kontenera. Wskaźniki nie zapewniają takiej ochrony.
- wielkość - iteratory mogą mieć różne rozmiary, w zależności od typu kontenera, w przeciwieństwie do wskaźników, które zawsze mają tę samą wielkość.
- przenośność - iteratory są bardziej przenośne niż wskaźniki. Na przykład, iterator zdefiniowany dla wektora jest przenośny, podczas gdy wskaźnik do elementu wektora może przestać działać po przeniesieniu wektora w inne miejsce w pamięci.

Iterator a wskaźnik

Podsumowując, iteratory i wskaźniki są podobne, ale iteratory oferują dodatkowe funkcjonalności, które umożliwiają łatwiejsze i bezpieczniejsze przeglądanie elementów w kontenerach.

Omówienie wybranych funkcji iteratorów

`std::advance` i `std::prev`

Funkcje te pozwalają na przesuwanie iteratora o określoną liczbę kroków. Funkcja `std::advance` zwraca iterator przesunięty w przód, a funkcja `std::prev` w tył.

```
std::vector<int> v = {1, 2, 3, 4, 5};
auto it = v.begin();
std::advance(it, 2);
std::cout << *it << std::endl;
it = std::prev(it, 1);
std::cout << *it << std::endl;
```

Omówienie wybranych funkcji iteratorów

std::distance

Funkcja pozwala na obliczenie odległości między dwoma iteratorami. Funkcja ta akceptuje dwa argumenty (iterator początkowy i końcowy), następnie zwraca liczbę kroków między nimi.

```
std::vector<int> v = {1, 2, 3, 4, 5};
auto it1 = v.begin();
auto it2 = v.end();
std::cout << std::distance(it1, it2) << std::endl;
```

Omówienie wybranych funkcji iteratorów

std::copy

Funkcja pozwala na kopiowanie elementów z jednego kontenera do drugiego. Funkcja ta akceptuje trzy argumenty: iterator początkowy źródłowego kontenera, iterator końcowy źródłowego kontenera oraz iterator początkowy docelowego kontenera.

```
std::vector<int> v1 = {1, 2, 3, 4, 5};
std::vector<int> v2(5);
std::copy(v1.begin(), v1.end(), v2.begin());
for (auto i : v2) {
    std::cout << i << "␣";
}
```

Omówienie wybranych funkcji iteratorów

std::transform

Funkcja pozwala na wykonanie określonej operacji na każdym elemencie z danego zakresu iteratorów i umieszczenie wyników w innym kontenerze. Wymaga ona podania trzech zakresów iteratorów: pierwszego, który wskazuje na początek wejściowego zakresu, drugiego, który wskazuje na koniec wejściowego zakresu, oraz trzeciego, który wskazuje na początek wyjściowego zakresu.

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    std::vector<int> input {1, 2, 3, 4, 5};
    std::vector<int> output(input.size());

    std::transform(input.begin(), input.end(), output.begin(),
                   [](int x) { return x * x; });

    for (int x : output) {
        std::cout << x << "␣";
    }
    std::cout << std::endl;

    return 0;
}
```

Praktyczne zastosowanie iteratorów w C++

Przetwarzanie kolekcji

```cpp
#include <iostream>
#include <vector>
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    for (auto it = numbers.begin(); it != numbers.end(); ) {
        if (*it % 2 == 0) {
            it = numbers.erase(it);
        } else {
            ++it;
        }
    }
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Praktyczne zastosowanie iteratorów w C++

Sortowanie i wyszukiwanie

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
    std::vector<int> numbers = {5, 3, 1, 4, 2};
    std::sort(numbers.begin(), numbers.end());
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    auto it = std::find(numbers.begin(), numbers.end(), 3);
    if (it != numbers.end()) {
        std::cout << "Element found: " << *it << std::endl;
    } else {
        std::cout << "Element not found" << std::endl;
    }
    return 0;
}
```

Praktyczne zastosowanie iteratorów w C++

Odczyt i zapis danych

```
#include <iostream>
#include <fstream>
#include <vector>
int main() {
    std::ifstream file("data.txt");
    std::vector<int> numbers;
    std::istream_iterator<int> it(file);
    std::istream_iterator<int> end;
    while (it != end) {
        numbers.push_back(*it++);
    }
    for (auto it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << "␣";
    }
```

Praktyczne zastosowanie iteratorów w C++

Przeszukiwanie drzewa w głąb - węzeł

```
#include <iostream>
#include <memory>

struct Node {
  int value;
  std::shared_ptr<Node> left;
  std::shared_ptr<Node> right;

  Node(int v, std::shared_ptr<Node> l, std::shared_ptr<Node> r) :
    value(v), left(l), right(r) {}
};
```

```
void traverse_preorder(std::shared_ptr<Node> node) {
  if (!node) {
    return;
  }

  std::cout << node->value << "␣";
  traverse_preorder(node->left);
  traverse_preorder(node->right);
}
```

Praktyczne zastosowanie iteratorów w C++

Przeszukiwanie drzewa w głąb - iteracyjne

```
void traverse_preorder_with_iterators(std::shared_ptr<Node> node)
  std::stack<std::shared_ptr<Node>> stack;
  stack.push(node);

  while (!stack.empty()) {
    auto current_node = stack.top();
    stack.pop();

    if (current_node) {
      std::cout << current_node->value << "␣";
      stack.push(current_node->right);
      stack.push(current_node->left);
    }
  }
}
```

Praktyczne zastosowanie iteratorów w C++

Przeszukiwanie drzewa w głąb - wywołanie

```
int main() {
    auto root = std::make_shared<Node>(5,
        std::make_shared<Node>(3,
            std::make_shared<Node>(1, nullptr, nullptr),
            std::make_shared<Node>(4, nullptr, nullptr)),
        std::make_shared<Node>(8, nullptr,
            std::make_shared<Node>(10, nullptr, nullptr)));

    traverse_preorder(root);
    traverse_preorder_with_iterators(root);
}
```

Częste problemy związane z iteratorami

- niezainicjowane iteratory - używanie iteratorów, które nie zostały zainicjowane, może prowadzić do nieprzewidywalnego zachowania programu.
- nieprawidłowe poruszanie się po kontenerze - błędne użycie operatorów inkrementacji/dekrementacji lub przesunięcia może prowadzić do wyjścia poza zakres kontenera.
- modyfikacja kontenera podczas iteracji - dodanie lub usunięcie elementu z kontenera podczas iteracji może spowodować nieprawidłowe zachowanie iteratorów.
- wykorzystanie nieprawidłowego iteratora - próba użycia iteratora z jednego kontenera do poruszania się po innym kontenerze może spowodować nieprzewidywalne zachowanie programu.
- brak wsparcia dla iteratorów - niektóre algorytmy mogą nie być dostępne dla niektórych typów iteratorów lub kontenerów.

Podsumowanie

- Iterator to obiekt pozwalający na poruszanie się po elementach kontenera.
- Nieprawidłowe użycie iteratorów może prowadzić do poważnych problemów związanych z pamięcią, takich jak dereferencja niezainicjalizowanego iteratora lub wyjście poza zakres kontenera.
- Podczas korzystania z iteratorów warto pamiętać o funkcjach z biblioteki standardowej, takich jak `std::find`, `std::for_each` czy `std::transform`, które ułatwiają przetwarzanie kontenerów za pomocą iteratorów.
- Prawidłowe korzystanie z iteratorów pozwala na szybkie i efektywne przetwarzanie danych w kontenerach.