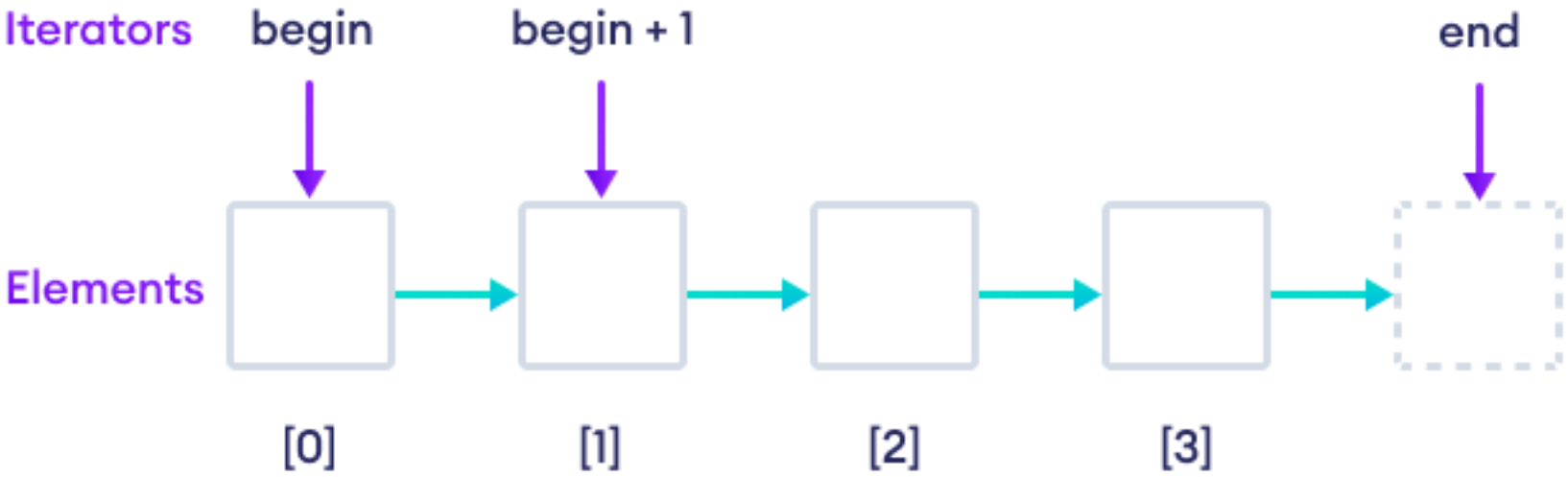
Итератори в C++

Какво е Итератор?

- Итераторът е обект, който сочи към елемент в контейнер
- Позволява последователно обхождане на елементите
- Работи като "умен указател" с допълнителна функционалност



Защо Използваме Итератори?

Проблеми:

- Различен код за обхождане при различни контейнери

Пример 1: Обхождане на масив

```cpp
int arr[] = {10, 20, 30, 40, 50};
int size = 5;

// За масив използваме индекси
for (int i = 0; i < size; i++) {
    cout << arr[i] << " ";
}
```

Пример 2: Обхождане на списък (linked list)

```cpp
struct Node {
    int data;
    Node* next;
};

Node* head = /* ... */;

// За списък трябва да използваме указатели
Node* current = head;
while (current != nullptr) {
    cout << current->data << " ";
    current = current->next;
}
```

Защо Използваме Итератори?

Проблеми:

- Различен код за обхождане при различни контейнери

Решението с Итератор:

- Един интерфейс за всички контейнери

```
// Шаблонна функция, която работи с ВСЕКИ контейнер
template<typename Container>
void printContainer(const Container& container) {
    // Един и същи код!
    for (auto it = container.begin(); it != container.end(); ++it) {
        cout << *it << " ";
    }
    cout << "\n";
}
```

Защо Използваме Итератори?

Проблеми:

- Различен код за обхождане при различни контейнери
- Необходимо е да знаем вътрешната структура на контейнера
- Кода е по-труден за поддръжка

Решението с Итератор:

- Един интерфейс за всички контейнери
- Скрива детайли за имплементацията
- Лесно преизползване на код

Основни Операции на Итератор

Методи на контейнера:

- being() – итератор към първия елемент
- end() – итератор „един след последния елемент“

Трите основни неща, които итераторът прави:

- Дереференция (*it) – дава стойността на елемента
- Инкрементиране (++it) – премества се към следващия елемент
- Сравнение (==, !=) – проверява позицията

Основни Операции на Итератор

Пример – Масив

```
int arr[] = {10, 20, 30, 40, 50};

// begin() - указател към първия елемент
int* begin = arr;  // или &arr[0]

// end() - указател "един след последния"
int* end = arr + 5;  // или &arr[5]
```

```
cout << "=== Обхождане на масив с итератор ===\n\n";

// Обхождане с итератор (в случая - указател)
for (int* it = begin; it != end; ++it) {
    // ОПЕРАЦИЯ 1: Сравнение (!=)
    // Проверяваме дали сме стигнали края

    // ОПЕРАЦИЯ 2: Дереферииране (*)
    // Получаваме стойността на елемента
    cout << "Адрес: " << it << " -> Стойност: " << *it << "\n";

    // ОПЕРАЦИЯ 3: Инкрементиране (++)
    // Премества се към следващия елемент
}
```

Основни Операции на Итератор

Пример – Динамичен Масив

```
vector<string> cities = {"Sofia", "Plovdiv", "Varna", "Burgas"};

cout << "=== Обхождане на vector с итератор ===\n\n";

// begin() - итератор към първия елемент
// end() - итератор "един след последния"
for (auto it = cities.begin(); it != cities.end(); ++it) {
    // ОПЕРАЦИЯ 1: Сравнение (!=)
    // ОПЕРАЦИЯ 2: Дереферииране (*)
    // ОПЕРАЦИЯ 3: Инкрементиране (++)
    cout << *it << "\n";
}
```

Основни Операции на Итератор

Методи на контейнера:

- being() – итератор към първия елемент
- end() – итератор „един след последния елемент“

Трите основни неща, които итераторът прави:

- Дереференция (*it) – дава стойността на елемента
- Инкрементиране (++it) – премества се към следващия елемент
- Сравнение (==, !=) – проверява позицията

Начини за Обхождане в C++

1. Класически for (с индекс):

```cpp
for (size_t i = 0; i < vec.size(); i++) {
    cout << vec[i] << " ";
}
```

2. For с итератор:

```cpp
for (auto it = vec.begin(); it != vec.end(); ++it) {
    cout << *it << " ";
}
```

3. Range-based for (C++11):

```cpp
for (const auto& element : vec) {
    cout << element << " ";
}
```

Ключовата Дума “auto” при Итераторите

Предимства на auto **при итераторите:**

- Компилаторът автоматично определя типа
- Код е по-четим и кратък
- Спестява писане при сложни типове

С `auto` (кратко):

```cpp
auto it = vec.begin();
```

Без `auto` (дълго):

```cpp
std::vector<int>::iterator it = vec.begin();
```

Промяна на Елемента при Обхождане

```
int main() {
    int arr[] ={[0]=1, [1]=2, [2]=3, [3]=4, [4]=5};

    printArray(begin: arr, end: arr + 5);

    for (int i = 0; i < 5; i++) {
      arr[i] = arr[i] * 10;
    }

    printArray(begin: arr, end: arr + 5);

    return 0;
}
```

Елементите се променят

```
1 2 3 4 5
10 20 30 40 50
```

Промяна на Елемента при Обхождане

```
int main() {
    std::vector<int> myVector = {[0]=1, [1]=2, [2]=3, [3]=4, [4]=5};

    printArray(begin: myVector.begin(), end: myVector.end());

    for (std::vector<int>::iterator it = myVector.begin(); it != myVector.end(); ++it) {
        *it *= 10;
    }

    printArray(begin: myVector.begin(), end: myVector.end());

    return 0;
}
```

Елементите се променят

```
1 2 3 4 5
10 20 30 40 50
```

Промяна на Елемента при Обхождане

```
int main() {
    std::vector<int> myVector = {[0]=1, [1]=2, [2]=3, [3]=4, [4]=5};

    printArray(begin: myVector.begin(), end: myVector.end());

    for (auto value: int: myVector) {
        value *= 10;
    }

    printArray(begin: myVector.begin(), end: myVector.end());

  return 0;
}
```

Елементите НЕ се промен

```
1 2 3 4 5
1 2 3 4 5
```

Промяна на Елемента при Обхождане

```
int main() {
    std::vector<int> myVector = {[0]=1, [1]=2, [2]=3, [3]=4, [4]=5};

    printArray(begin: myVector.begin(), end: myVector.end());

    for (auto& value: int &: myVector) {
        value *= 10;
    }

    printArray(begin: myVector.begin(), end: myVector.end());

    return 0;
}
```

Елементите се променят

```
1 2 3 4 5
10 20 30 40 50
```

Промяна на Елемента при Обхождане

```
int main() {
    std::vector<int> myVector = {[0]=1, [1]=2, [2]=3, [3]=4, [4]=5};

    printArray(begin: myVector.begin(), end: myVector.end());

    for (const auto& value: int const &: myVector) {
        value *= 10;
    }

    printArray(begin: myVector.begin(), end: myVector.end());

    return 0;
}
```

Грешка при Компилация

Промяна на Елемента при Обхождане

```cpp
vector<int> numbers = {1, 2, 3, 4, 5};

// Копие - НЕ променя оригинала
for (auto num : numbers) {
    num = num * 2;  // Промяна само в копието
}

// Референция - ПРОМЕНЯ оригинала
for (auto& num : numbers) {
    num = num * 2;  // Променя реалните стойности
}

// Константна референция - ефективно четене
for (const auto& num : numbers) {
    cout << num << " ";  // Без копиране
}
```

Промяна на Елемента при Обхождане

Начин	**Синтаксис**	**Променя оригинала?**	**Копира елементите?**	**Кога да използваме?**
Копие	`for (auto x : vec)`	✗ НЕ	✓ ДА	Когато искаме локална копия и няма да променяме
Референция	`for (auto& x : vec)`	✓ ДА	✗ НЕ	Когато искаме да ПРОМЕНИМ оригинала
Const референция	`for (const auto& x : vec)`	✗ НЕ	✗ НЕ	Когато САМО ЧЕТЕМ (най-ефективно!)

Итератор Design Pattern

Методи на контейнера:

- being() – итератор към първия елемент
- end() – итератор „един след последния елемент“

Връщат

Итератор Клас

Имплементира

Трите основни неща, които итераторът прави:

- Дереференция (*it) – дава стойността на елемента
- Инкрементиране (++it) – премества се към следващия елемент
- Сравнение (==, !=) – проверява позицията

Итератор Design Pattern – Пример

```
public:
    class Iterator {
    private:
        Node* current;
    public:
        Iterator(Node* node) : current(node) {}

        T& operator*() { return current->data; }

        Iterator& operator++() {
            current = current->next;
            return *this;
        }

        bool operator!=(const Iterator& other) {
            return current != other.current;
        }
    };

    Iterator begin() { return Iterator(head); }
    Iterator end() { return Iterator(nullptr); }
};
```

Итератор Design Pattern – Задача

Задача: Напишете итератор за свой списък, който:

- Има operator*, operator++, operator!=
- Работи с begin() и end()
- Може да се използва в range-based fo

Бонус: Добавете reverse итератор (rbeg

Готови Имплементации на Контейнери

Singly Linked List
https://www.geeksforgeeks.org/cpp/program-to-implement-singly-linked-list-

Vector
https://www.geeksforgeeks.org/cpp/how-to-implement-our-own-vector-class-in

Итератор Design Pattern – Чести Грешки

❌ Грешка 1: Модификация при обхождане

```cpp
for (auto it = vec.begin(); it != vec.end(); ++it) {
    vec.push_back(10);  // Инвалидира итератора!
}
```

❌ Грешка 2: Дереферииране на `end()`

```cpp
auto it = vec.end();
cout << *it;  // Грешка - end() е след последния елемент!
```

✅ Правилно:

```cpp
if (it != vec.end()) {
    cout << *it;
}
```

Промяна на Елемента при Обхождане

```cpp
vector<int> numbers = {1, 2, 3, 4, 5};

// Копие - НЕ променя оригинала
for (auto num : numbers) {
    num = num * 2;  // Промяна само в копието
}

// Референция - ПРОМЕНЯ оригинала
for (auto& num : numbers) {
    num = num * 2;  // Променя реалните стойности
}

// Константна референция - ефективно четене
for (const auto& num : numbers) {
    cout << num << " ";  // Без копиране
}
```

Предимства на Итераторите

Абстракция:
- Скриват имплементационни детайли
- Един интерфейс за различни структур

Гъвкавост:
- Множество итератори върху един контейнер
- Различни начини на обхождане

Съвместимост:
- Работят с STL алгоритми (sort, find copy)
- Стандартен интерфейс

Proxy Pattern в C++ Homework

https://www.geeksforgeeks.org/system-design/proxy-pattern-c-design-patterns/

Източници

- https://www.w3schools.com/cpp/cpp_iterators.asp
- https://www.geeksforgeeks.org/cpp/introduction-iterators-c/
- https://courses.grainger.illinois.edu/cs225/fa2022/resources/iterators/
- https://refactoring.guru/design-patterns/iterator
- https://stackoverflow.com/questions/27583945/c-iterator-vector-struct
- https://www.learncpp.com/cpp-tutorial/introduction-to-iterators/
- https://www.cs.up.ac.za/cs/lmarshall/TDP/Notes/_Chapter15_Iterator.pdf
- https://sourcemaking.com/design_patterns/iterator
- https://www.programiz.com/cpp-programming/iterators
- https://dotnettutorials.net/lesson/iterators-in-cpp/