Итераторы — конспект темы

Понятие итератора

Итератор — объект, для которого определены некоторые действия. Он не содержит данных и может только указывать на них.

Итератор умеет показывать на что-нибудь, что элементом не будет. В отличие от ссылки, итератор можно двигать.

Итераторы защищают контейнер от неожиданных изменений. В алгоритме `count` методы `xs.begin()` и `xs.end()` возвращают итераторы, и дальше алгоритм работает с итераторами:

```
const vector<int> xs = {1, 2, 1, 1, 5};
cout << count(xs.begin(), xs.end(), 1) << endl; // 3
```

Через итераторы алгоритм получает доступ к элементам контейнера, может итерировать по элементам, но к самому контейнеру у алгоритма доступа нет. Вектор остаётся неизменным. Не меняется порядок элементов, размер не увеличивается и не уменьшается. Сделать это через итераторы нельзя. А заменить вектор на другой контейнер — можно.

Концепция полуинтервалов

Что-либо называется **интервалом**, если начало и конец НЕ входят в число элементов:

(1; 10) — здесь и 1, и 10 НЕ входят в интервал. То есть входят в него целые числа 2, 3, 4, 5, 6, 7, 8 и 9.

Что-либо называется полуинтервалом, если что-нибудь одно (начало или конец) не входит в число элементов:

[1; 10) — здесь 1 будет элементом, а 10 уже нет. В полуинтервал входят целые числа 1, 2, 3, 4, 5, 6, 7, 8, и 9.

Под концом итератора может подразумеваться:

- последний элемент контейнера;

- место, не содержащее никакого элемента и находящееся за последним из них.

Итераторы в конструкторах контейнеров

Итераторы удобны в конструкторах. Из вектора языков с повторениями легко сделать множество уникальных языков:

```
#include <algorithm>
#include <iostream>
#include <set>
#include <string>
#include <vector>

using namespace std;

template <typename It>
void PrintRange(It range_begin, It range_end) {
    for (auto it = range_begin; it != range_end; ++it) {
        cout << *it << " "s;
    }
    cout << endl;
}

template <typename It>
auto MakeSet(It range_begin, It range_end) {
    return set(range_begin, range_end);
}

int main() {
    vector<string> langs = {"Python"s, "Java"s, "C#"s, "Ruby"s, "C++"s, "C++"s, "C++"s, "R
uby"s, "Java"s};
    auto unique_langs = MakeSet(langs.begin(), langs.end());
    PrintRange(unique_langs.begin(), unique_langs.end());
}
```

Итераторы в методах контейнеров

Прямое назначение итераторов — перебирать элементы контейнера. Зная место в контейнере, можно удалить элемент или вставить новый. Итераторы позволяют не думать о порядковом номере элемента.

Метод `insert` вставляет элемент в определённое место в контейнере — перед итератором, который передан в качестве параметра.

```
#include <algorithm>
#include <iostream>
```

```
#include <string>
#include <vector>

using namespace std;

template <typename It>
void PrintRange(It range_begin, It range_end) {
    for (auto it = range_begin; it != range_end; ++it) {
        cout << *it << " "s;
    }
    cout << endl;
}

int main() {
    vector<string> langs = {"Python"s, "Java"s, "C#"s, "Ruby"s, "C++"s};
    auto it = find(langs.begin(), langs.end(), "C++"s);
    langs.insert(it, "C"s);
    PrintRange(langs.begin(), langs.end());
}
```

Аналогично `insert` действует метод `erase`.

Если контейнер был изменён, возможно, ранее использованные итераторы инвалидированы. Работать с ними уже нельзя.

Итератор не знает, что с контейнером что-то произошло, и не может модифицировать сам себя. За это отвечает пользователь.

Категории итераторов

Типы итераторов отличаются по набору поддерживаемых действий.

- **Итератор чтения (Input iterator).** Гарантирует возможность чтения элемента, но не гарантирует запись. Используется в алгоритме `find`.
- **Итератор записи (Output iterator).** Гарантирует запись элемента, но не гарантирует чтение. Используется в алгоритме copy.
- **Однонаправленный итератор (Forward iterator).** Итератор чтения с возможностью обойти элементы контейнера, но только в одном направлении. Используется в алгоритме `replace`.
- **Двунаправленный итератор (Bidirectional iterator).** Однонаправленный итератор, умеющий ходить назад. Используется в алгоритме reverse.
- **Итератор произвольного доступа (Random access iterator).** Поддерживает те же действия, что двунаправленный итератор, а также:

- операции сложения и вычитания с числами для перемещения сразу на определённое расстояние от текущей позиции;
- возможность вычитать итераторы друг из друга;
- операторы сравнения `>`, `<`, `>=`, `<=`;
- оператор `[ ]`.

Используется в алгоритме `random_shuffle`.

Стандартные алгоритмы — рекурсия

Рекурсия значит, что в тот момент, когда ваш алгоритм должен начать повторять одно и то же действие, он вызывает сам себя.

По сути, рекурсия — это цикл, только действия цикла находятся в функции, которая на каждом шаге вызывает саму себя.

Вычисление факториала циклом:

```
uint64_t Factorial(int num) {
    int factorial = 1;
    while (num > 1) {
        factorial *= num;
        --num;
    }
    return factorial;
```

Вычисление факториала рекурсией:

```
uint64_t Factorial(int num) {
    int factorial = 1;
    if (0 != num) {
        factorial = Factorial(num - 1) * num;
    }
    return factorial;
}
```

Перед тем, как использовать в коде рекурсию, спланируйте её. Ответьте на вопросы:

1. Какие параметры она будет принимать? Часто верные параметры — уже половина решения.

2. Какие действия должна совершить функция на каждом шаге?
3. Каково условие завершения?