Junior C++ developer

Лекция 10

- Итераторы
- Лямбда-функции

Контейнеры STL

- Последовательные контейнеры
- Ассоциативные контейнеры
- Адаптеры

Последовательные контейнеры

- **std::array** (начиная с C++11) - статический непрерывный массив
- **std::vector -** динамический непрерывный массив
- **std::deque** - двусторонняя очередь
- **std::forward_list -** односвязный список
- **std::list** - двусвязный список

Ассоциативные контейнеры

- **std::set -** коллекция уникальных ключей, отсортированная по ключам
- **std::map** - коллекция пар ключ-значение, отсортированная по ключам, ключи являются уникальными
- **std::multiset** - коллекция ключей, отсортированная по ключам
- **std::multimap -** коллекция пар ключ-значение, отсортированная по ключам

Неупорядоченные ассоциативные контейнеры

- **std::unordered_set** (начиная с C++11) - коллекция уникальных ключей, хэш-ключами
- **std::unordered_map** (начиная с C++11) коллекция пар ключ-значение, хэширован ключи, ключи являются уникальными
- **std::unordered_multiset** (начиная с C++11) - коллекция ключей, хэш-ключами
- **std::unordered_multimap** (начиная с C++11) - коллекция пар ключ-значение, хешируется по ключам

Адаптеры

- **std::stack -** адаптируется контейнеров обеспечить стек (LIFO структуры данных)
- **std::queue -** адаптируется контейнера обеспечивают очереди (FIFO структуры данных)
- **std::priority_queue -** адаптация контейнеров для обеспечения приоритета очереди

Итераторы

Итератор — это объект, который способен перебирать элементы **класса-контейнера** без необходимости пользователю знать, как реализован определенный контейнерный класс. Во многих контейнерах (особенно в списке и ассоциативных контейнерах) итераторы являются основным способом доступа к элементам этих контейнеров.

Операторы итераторов

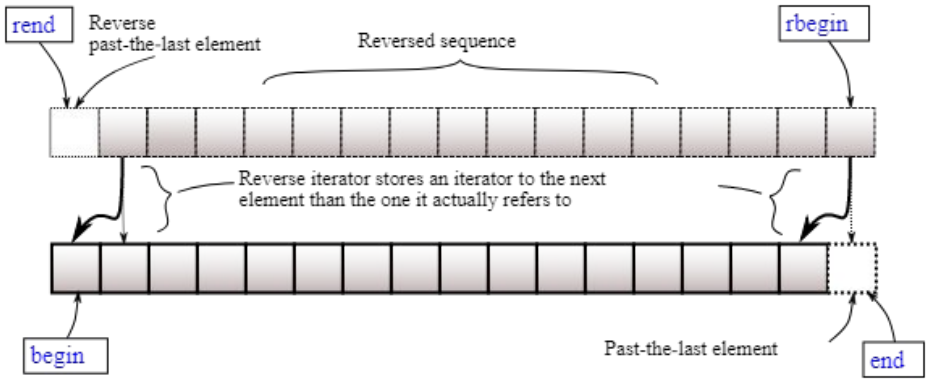
- ***operator**** возвращает элемент, на который в данный момент указывает итератор.
- ***operator++*** перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют ***operator--***.
- ***operator== и operator!=*** используются для определения того, указывают ли два итератора на один и тот же элемент или нет. Для сравнения значений, на которые указывают два итератора, нужно сначала разыменовать эти итераторы.
- ***operator=*** присваивает итератору новую позицию (обычно начало или конец элементов контейнера). Чтобы присвоить значение элемента, на который указывает итератор, другому объекту, нужно сначала разыменовать итератор.

Основные методы итераторов

- ***std::begin*** - возвращает итератор на начало последовательности
- ***std::cbegin*** - возвращает константный итератор на начало последовательности
- ***std::end()*** - возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере.
- ***std::cend()*** - возвращает константный итератор, представляющий элемент, который находится после последнего элемента в контейнере.

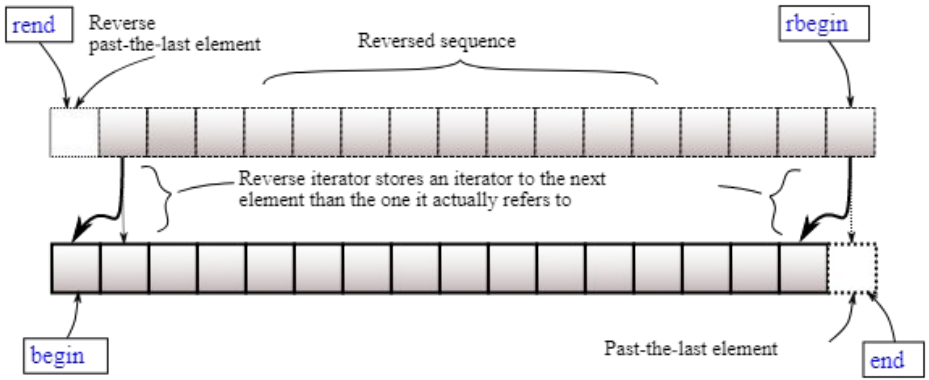
Основные методы итераторов

- ***std::rbegin*** - возвращает итератор на конец последовательности
- ***std::crbegin*** - возвращает константный итератор на конец последовательности



Основные методы итераторов

- ***std::rend*** - возвращает итератор на элемент перед первым элементом
- ***std::crend*** - возвращает константный итератор на элемент перед первым элементом



Лямбда

Лямбда-выражениями называются безымянные локальные функции, которые можно создавать прямо внутри какого-либо выражения.

```
[captures](arg1, arg2) -> result_type {  /* code */  }
```

- ***arg1, arg2*** - это аргументы. То, что передается алгоритмом в лямбду.
- ***result_type*** - это тип возвращаемого значения. Это может показаться несколько непривычно, так как раньше тип всегда писали перед сущностью (переменной, функцией).
- ***captures*** - список захвата: переменных внешней среды, которые стоит сделать доступными внутри лямбды. Эти переменные можно захватывать по значению и по ссылке.

Практика

Есть некоторый целочисленный вектор. Задача состоит в том, чтобы отсортировать элементы так, чтобы слева находились нечетные элементы, а справа — четные.

Решение

```
bool Comparator(int lhs, int rhs)
{
    if (lhs & 1  &&  rhs & 1)
        return lhs < rhs;
    return lhs & 1;
}

int main()
{
    vector<int> vec = {1,3, 4, 5, 6, 8, 10};
    std::sort(vec.begin(), vec.end(), Comparator);

    for(auto i : vec)
    {
        cout << i << " ";
    }
}
```

Решение

```
bool Comparator(int lhs, int rhs)
{
    if (lhs & 1  &&  rhs & 1)
        return lhs < rhs;
    return lhs & 1;
}

int main()
{
    vector<int> vec = {1,3, 4, 5, 6, 8, 10};
    std::sort(vec.begin(), vec.end(), Comparator);

    for(auto i : vec)
    {
        cout << i << " ";
    }
}
```

Недостатки решения - реализация функции, которая используется только один раз

Решение с помощью лямбда

```
int main()
{
    vector<int> vec = {1,3, 4, 5, 6, 8, 10};
    std::sort(vec.begin(), vec.end(), [](int lhs, int rhs) -> bool {
        if (lhs & 1  &&  rhs & 1)
            return lhs < rhs;
        return lhs & 1;
    });
}
```

Если лямбда состоит из одного оператора return, то возвращаемый тип можно не писать

```
std::sort(vec.begin(), vec.end(), [](int lhs, int rhs)
{
    return lhs & 1;
});
```

Передача по ссылке и по значению

```
int max = 4;

// по значению
std::sort(vec.begin(), vec.end(), [max](int lhs, int rhs) {
    return lhs < max;
});


// по ссылке
std::sort(vec.begin(), vec.end(), [&max](int lhs, int rhs) {
    return lhs < max;
});
```

Захват всех переменных области видимости

```
// по значению
std::sort(vec.begin(), vec.end(), [=](int lhs, int rhs) {
    return lhs < someVar;
});

// по ссылке
std::sort(vec.begin(), vec.end(), [&](int lhs, int rhs) {
    return lhs < otherVar;
});
```

Присвоение переменным

```
auto square = [](int x) { return x * x; };
std::cout << square(16) << std::endl;
```

Обращение к полю класса

```
class Foo
{
public:
    Foo(): _x(5) {}

    void doSomething() {
        // если вместо this поставить _x — будет ошибка!
        auto lambda = [this](int x) {
            std::cout << _x * x << std::endl;
        };

        lambda(4);
    }

private:
    int _x;
};
```

Практика

Для вектора строк вывести все строки начинающиеся с символа ‘C’.

Для поиска использовать алгоритм ***std::find_if***

```
int main()
{
    vector<string> strs = {"Cstr1", "Astr1", "Bstr1", "Cstr2", "Astr2", "Cstr3"};
    auto result = find_if(strs.begin(), strs.end(), [](const string& s)
    {
        return s[0] == 'C';
    });

    cout << *result;

}
```

лямбда

***result** - ссылка на объект в последовательности. Причем ссылка не константная, то есть с ее помощью мы можем изменить элемент последовательности

Практика

Добавим проверку для случая когда элементов в последовательности не найдено

```
int main()
{
    vector<string> strs = {"Cstr1", "Astr1", "Bstr1", "Cstr2", "Astr2", "Cstr3"};
    auto result = find_if(strs.begin(), strs.end(), [](const string& s)
    {
        return s[0] == 'C';
    });

    if(result == strs.end())
    {
        cout << "no elements";
    }
    else
    {
        cout << *result;
    }
}
```

Проверка на пустой результат

Вопрос

Что будет если обратиться к элементу последовательности с помощью итератора ***end()***?

Вопрос

Что будет если обратиться к элементу последовательности с помощью итератора ***end()***?

Итераторы работают по принципу полуинтервалов

[begin(), end()) - begin() включительно, end() не включительно.

end() указывает на элемент следующий за последним.

Обращаться к элементу по ***end()*** нельзя.

Синонимы

strs.begin() и ***begin(strs)***

strs.end() и ***end(strs)***

Итераторы для std::map

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

    cout << endl;
}
```

```
int main()
{
    map<string, int> m_langs = {
            {"Python", 27},
            {"C++", 35},
            {"C", 46},
            {"Java", 23},
            {"C#", 18},
            {"Js", 23}
    };

    printRange(m_langs.begin(),
m_langs.end());
}
```

Итераторы для std::map

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

    cout << endl;
}
```

Данный код не скомпилируется, так как элементом std::map является std::pair для которого operator<< не определен

```
int main()
{
    map<string, int> m_langs = {
            {"Python", 27},
            {"C++", 35},
            {"C", 46},
            {"Java", 23},
            {"C#", 18},
            {"Js", 23}
    };

    printRange(m_langs.begin(),
m_langs.end());
}
```

Итераторы для std::map

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

    cout << endl;
}
```

Данный код не скомпилируется, так как элементом std::map является std::pair для которого operator<< не определен

```
int main()
{
    map<string, int> m_langs = {
            {"Python", 27},
            {"C++", 35},
            {"C", 46},
            {"Java", 23},
            {"C#", 18},
            {"Js", 23}
    };

    printRange(m_langs.begin(),
m_langs.end());
}
```

Варианты решения проблемы:

- Реализовать еще одну функцию для std::map
- Реализовать специализацию шаблона для std::map

Специализация шаблона

```
template <>
void printRange<map<string, int>::iterator>(map<string, int>::iterator range_begin,
map<string, int>::iterator range_end)
{
    for (auto it = range_begin; it != range_end; it++)
    {
        cout << it->first << " " << it->second << endl;
    }

    cout << endl;
}
```

Ключевое слово using

```
using map_it = map<string, int>::iterator;

template <>
void printRange<map_it>(map_it range_begin,map_it range_end)
{
    for (auto it = range_begin; it != range_end; it++)
    {
        cout << it->first << " " << it->second << endl;
    }

    cout << endl;
}
```

Опасные операции с итераторами

*end(element)

auto it = end(element); ++it;

auto it = begin(element); --it;

Отличия итераторов от ссылок

- Итераторы могут указывать в никуда - на end
- Итераторы можно перемещать на другие элементы