Семинар #6: Итераторы и алгоритмы.

Алгоритмы

Библиотека `algorithm`

Библиотека `algorithm` предоставляет шаблонные функции, которые реализуют множество различных алгоритмов для работы с набором элементов, хранящимся в контейнере. Как правило все эти функции работаю с диапазоном элементов, который задаётся двумя итераторами: итератором на первый элемент диапазона и итератором на элемент, следующим за последним элементом диапазона.

Функции `min_element` и `max_element`

Шаблонные функции `min_element` и `max_element` находят минимальный и максимальный элемент на некотором диапазоне, задаваемом итераторами. Функции возвращают итератор на этот элемент.

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

int main()
{
    std::vector<int> v {40, 20, 50, 30, 10};
    std::cout << *std::min_element(v.begin(), v.end()) << std::endl;        // 10
    std::cout << *std::min_element(v.begin(), v.begin() + 3) << std::endl;  // 20

    std::list<std::string> l {"Tiger", "Lion", "Axolotl"};
    std::cout << *std::min_element(l.begin(), l.end()) << std::endl;        // Axolotl
}
```

При этом элемент контейнера должен иметь оператор `<` для сравнения объектами того же типа. Если такого оператора у типа контейнера нет, то необходимо передать компаратор третьим аргументом в функции. Если этого не сделать, то произойдёт ошибка компиляции.

Функция `sort`

Шаблонная функция `std::sort` сортирует диапазон элементов контейнера. Диапазон задаётся двумя итераторами: итератором на первый элемент и итератором на элемент, следующим за последним элементом.

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

int main()
{
    std::vector<int> v {40, 20, 50, 30, 10};
    std::sort(v.begin(), v.end());

    for (auto elem : v)
        std::cout << elem << " ";
    std::cout << std::endl;
}
```

Также как и для функций поиска минимума и максимума, элемент контейнера должен иметь оператор < для сравнения объектами того же типа. Ведь для того, чтобы отсортировать элементы, нужно уметь их сравнивать.

Но, в отличии от функций поиска минимума и максимума, функция `std::sort` налагает более строгие ограничения на итераторы. Дело в том, что `std::sort` реализована с помощью модернизированого алгоритма быстрой сортировки и при реализации такого алгоритма требуется доступаться к элементу диапазона по его индексу. Функции `std::sort` передаются на вход два итератора и, чтобы доступаться к элементу диапазона по его индексу, используется операция прибавления целого числа к итератору. Если у итератора нет такой операции, то функция сортировки работать не будет, произойдёт ошибка компиляции. Например, у итератора `std::list` нет операции сложения с целым числом, поэтому попытка отсортировать `std::list` с помощью `std::sort` приведёт к ошибке.

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

int main()
{
    std::list<int> a {40, 20, 50, 30, 10};
    std::sort(a.begin(), a.end());  // Ошибка, нельзя сортировать std::list с помощью std::sort

    for (auto elem : a)
        std::cout << elem << " ";
    std::cout << std::endl;
}
```

Этот пример показывает, что разные шаблонные функции могут накладывать разные ограничения на объекты с которыми они работают. При работе с такими функциями нужно следить за тем выполняются ли все условия, накладываемые этой шаблонной функцией.

Функция `reverse`

Шаблонная функция `std::reverse` обращает диапазон элементов.

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

int main()
{
    std::vector<int> a {10, 20, 30, 40, 50};
    std::reverse(a.begin(), a.end());

    for (auto elem : a)
        std::cout << elem << " ";  // Напечатает 50 40 30 20 10
    std::cout << std::endl;


    std::list<std::string> b {"Axolotl", "Bat", "Cat"};
    std::reverse(b.begin(), b.end());

    for (auto elem : b)
        std::cout << elem << " ";  // Напечатает Cat Bat Axolotl
    std::cout << std::endl;
}
```

Функция `copy`

Шаблонная функция `std::copy` принимает на вход три итератора. Первые два итератора задают множество элементов которые нужно скопировать, а третий – место куда нужно их скопировать. Причём типы первых двух итераторов должны совпадать, а тип третьего итератора может отличаться. То есть можно, например, скопировать элементы вектора в связный список или в другой контейнер.

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

int main()
{
    std::vector<int> a {10, 20, 30, 40, 50};   // Вектор из элементов 10 20 30 40 50
    std::vector<int> b(5);                     // Вектор из элементов 0 0 0 0 0
                                               //
    std::copy(a.begin(), a.end(), b.begin()); //
    for (auto elem : b)                        //
        std::cout << elem << " ";              // Напечатает 10 20 30 40 50
    std::cout << std::endl;                    //
                                               //
    std::list<int> c(5);                       // Связный список из элементов 0 0 0 0 0
    std::copy(a.begin(), a.end(), c.begin()); //
    for (auto elem : c)                        //
        std::cout << elem << " ";              // Напечатает 10 20 30 40 50
    std::cout << std::endl;                    //
}
```

При работе с функцией `std::copy` важно помнить, что эта функция сама по себе не добавляет элементы в контейнер, она просто изменяет уже существующие элементы. В том месте куда элементы копируются должно быть достаточно места под копируемые элементы. Если необходимого места не будет, то это неопределённое поведение.

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

int main()
{
    std::vector<int> a {10, 20, 30, 40, 50};   // Вектор из элементов 10 20 30 40 50
    std::vector<int> b;                        // Пустой вектор

    std::copy(a.begin(), a.end(), b.begin()); // В векторе b нет места под 5 элементов
                                               // Это ошибка - неопределённое поведение
}
```

Функция `count`

Шаблонная функция `std::count` подсчитывает количество элементов в диапазоне, равных данному.

```
std::vector<int> a {50, 10, 20, 10, 20, 40, 10, 10, 30};
std::cout << std::count(a.begin(), a.end(), 10) << std::endl;    // Напечатает 4
std::cout << std::count(a.begin(), a.end(), 20) << std::endl;    // Напечатает 2
```

Функция `find`

Шаблонная функция `std::find` ищет первый элемент в диапазоне элементов, который равен данному. Если такого элемента в диапазоне нет, то функция возвращает итератор на элемент, следующий за последним элементом диапазона.

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

int main()
{
    std::vector<int> a {50, 10, 20, 10, 20, 40, 10, 10, 30};

    auto it1 = std::find(a.begin(), a.end(), 40);
    std::cout << (it1 - a.begin()) << std::endl;  // Напечатает 5

    auto it2 = std::find(a.begin(), a.end(), 20);
    std::cout << (it2 - a.begin()) << std::endl;  // Напечатает 2

    auto it3 = std::find(a.begin(), a.end(), 70);
    if (it3 == a.end())
        std::cout << "Element 70 not found" << std::endl;

    auto it4 = std::find(a.begin(), a.begin() + 3, 40);
    if (it4 == a.begin() + 3)
        std::cout << "Element 40 is not among the first three elements" << std::endl;
}
```

Функция `fill`

Шаблонная функция `std::fill` задаёт все элементы диапазона данным значением.

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

int main()
{
    std::vector<int> a {10, 20, 30, 40, 50}

    std::fill(a.begin(), a.end(), 70);

    for (auto elem : a)
        std::cout << elem << " ";  // Напечатает 70 70 70 70 70
    std::cout << std::endl;
}
```

Функция `remove` и метод `erase`

Шаблонная функция `std::remove` принимает на вход диапазон элементов, задаваемый двумя итераторами и некоторый элемент. Перемещает все элементы, неравные данному в начало диапазона. Остальные элементы диапазона могут получить произвольные значения. В результате диапазон как бы разделится на две части: первая часть состоит из элементов не равных данному, а вторая часть состоит из произвольных элементов. Функция возвращает итератор на первый элемент второй части. Порядок элементов в первой части диапазон будет тем же, каким он был в изначальном диапазоне.

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

int main()
{
    std::vector<int> a {50, 10, 20, 10, 20, 40, 10, 10, 30};
    auto border = std::remove(a.begin(), a.end(), 10);

    for (auto it = a.begin(); it != a.end(); ++it)
        std::cout << *it << " ";                    // Напечатает 50 20 20 40 30 40 10 10 30
    std::cout << std::endl;

    for (auto it = a.begin(); it != border; ++it)
        std::cout << *it << " ";                    // Напечатает 50 20 20 40 30
    std::cout << std::endl;
}
```

Шаблонная функция `std::remove` сама по себе не удаляет элементы, она просто перемещает элементы, не равные данному, в начало диапазона. Чтобы действительно удалить элементы из контейнера нужно воспользоваться методами контейнера. У разных контейнеров способ удаления может различаться, а у некоторых контейнеров (например, у `std::array`) удалить элементы вообще нельзя. Однако, большинство контейнеров поддерживают метод `erase`, который принимает два итератора и удаляет все элементы в диапазоне, задаваемом этими итераторами.

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

int main()
{
    std::vector<int> a {50, 10, 20, 10, 20, 40, 10, 10, 30};
    auto border = std::remove(a.begin(), a.end(), 10);
    a.erase(border, a.end());

    for (auto it = a.begin(); it != a.end(); ++it)
        std::cout << *it << " ";                    // Напечатает 50 20 20 40 30
    std::cout << std::endl;
}
```

Функция unique

Шаблонная функция `std::unique` "удаляет"повторяющиеся элементы в диапазоне. Работает по тому же принципу, что и функция `std::remove`. То есть, вместо удаления элементов, перемещает все нужные элементы в начало диапазона, а в оставшейся часть диапазона будут находиться произвольные значения. Возвращает итератор на первый элемент второй части диапазона.

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

int main()
{
    std::vector<int> a {20, 20, 10, 10, 10, 10, 20, 20, 30};
    auto border = std::unique(a.begin(), a.end());
    a.erase(border, a.end());

    for (auto it = a.begin(); it != a.end(); ++it)
        std::cout << *it << " ";                        // Напечатает 20 10 20 30
    std::cout << std::endl;
}
```

Библиотека numeric

Функция iota

Функция accumulate

Input и Output итераторы

Итератор `std::back_insert_iterator`

`std::back_insert_iterator<Container>` – это специальный итератор у которого операторы перегружены по другому. Для него:

- `operator*` ничего не делает
- `operator++` ничего не делает
- `operator=` вызывает метод `push_back` контейнера

Благодаря таким перегрузкам поведение этого итератора сильно отличается от поведения обычных итераторов. К примеру, следующий код добавит в контейнер `a` ещё один элемент:

```
std::vector<int> a { 1, 2, 3 };
std::back_insert_iterator<std::vector<int>> it{a};
*it = 4;
```

Так как тип этого итератора может иметь длинное название, то была введена функция под названием `std::back_inserter`, которая принимает на вход контейнер и возвращает такой итератор. Пример в котором вектор `a` копируется в пустой вектор `b` (полная версия в `std_copy_back_inserter.cpp`):

```
std::vector<int> a {10, 20, 30, 40, 50};
std::vector<int> b;
std::cout << b;

std::copy(a.begin(), a.end(), std::back_inserter(b));
std::cout << b;
```

- Напишите функцию `append`, которая будет принимать 2 вектора. Первый вектор эта функция должна принимать по ссылке, а второй – по константной ссылке. Функция должна копировать всё содержимое второго вектора в первый с помощью функции `std::copy`.

Итератор `std::ostream_iterator`

`std::ostream_iterator<T>` – это специальный итератор у которого операторы перегружены по другому:

- `operator*` ничего не делает
- `operator++` ничего не делает
- `operator=` выводит соответствующий элемент в выходной поток (например, `std::cout` или файл) с помощью оператора `<<`

- Что сделает эта программа:

```
int main() {
    std::ostream_iterator<int> it{ std::cout, ", " };
    it = 1;
    *it = 2;
    it++ = 3;
}
```

- Напишите программу, которая печатает числа от 1 до 100, разделённые символом `+`. Используйте `ostream_iterator`.
- Напишите программу, которая печатает содержимое вектора на экран, используя `std::copy` и `ostream_iterator`.
- Пусть есть такое множество строк:

```
std::set<std::string> a{"Cat", "Dog", "Mouse", "Elephant"};
```

Напечатайте содержимое этого множество на экран, каждый элемент на новой строке.

Категории итераторов

Функциональные объекты

Указатели на функции

Функторы

Функторы в стандартной библиотеке

Лямбда-функции

`std::function`

Указатели на методы, функция `std::mem_fn`

Стандартные алгоритмы с функциональными объектами

Функция `for_each`

Функция `find_if`

Функция `count_if`

Функции `all_of`, `any_of` **и** `none_of`

Функция `generate`

Функция `copy_if`

Функция `transform`

Функция `sort`

Функция `partition`

Функция `stable_partition`

Замыкания

Контейнеры

Стандартная библиотека включает в себя множество разных шаблонных контейнеров и алгоритмов для работы с ними.

контейнер	описание и основные свойства
`std::vector`	Динамический массив Все элементы лежат вплотную друг к другу, как в массиве Есть доступ по индексу за $O(1)$
`std::list`	Двусвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на элемент
`std::forward_list`	Односвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на предыдущий элемент
`std::set`	Реализация множества на основе сбалансированного дерева поиска Хранит элементы без дубликатов, в отсортированном виде Тип элементов должен реализовать `operator<` (или предоставить компаратор) Поиск/вставка/удаление элементов за $O(\log(N))$
`std::map`	Реализация словаря на основе сбалансированного дерева поиска Хранит пары ключ-значения без дубликатов ключей, в отсортированном виде Тип ключей должен реализовать `operator<` (или предоставить компаратор) Поиск/вставка/удаление элементов за $O(\log(N))$
`std::unordered_set`	Реализация множества на основе хеш-таблицы Хранит элементы без дубликатов, в произвольном порядке Поиск/вставка/удаление элементов за $O(1)$ в среднем
`std::unordered_map`	Реализация словаря на основе хеш-таблицы Хранит пары ключ-значения без дубликатов ключей,в произвольном порядке Поиск/вставка/удаление элементов за $O(1)$ в среднем
`std::multiset`	То же самое, что `std::set`, но может хранить дублированные значения
`std::deque`	Двухсторонняя очередь Добавление/удаление в начало и конец за $O(1)$

`std::stack`
`std::queue`
`std::priority_queue`
	Стек
	Очередь
	Очередь с приоритетом

`std::pair`	Пара элементов, могут быть объектами разных типов Элементы пары хранятся в публичных полях `first` и `second`
`std::tuple`	Фиксированное количество элементов, могут быть объектами разных типов
`std::array`	Массив фиксированного размера, все элементы имеют один тип

Итераторы

Итератор – это абстракция для итерирования по контейнеру. Многие контейнеры STL имеют вложенный тип `iterator`. Объекты этого типа используются для итерирования по контейнеру. Контейнеры имеют следующие методы:

- `begin()` – возвращает итератор на первый элемент
- `end()` – возвращает итератор на фиктивный элемент, следующий после последнего

С итератором можно проводить следующие операции:

`*it`: перегруженный `operator*` – возвращает ссылку на элемент, на который указывает итератор

`it++`: переходим к следующему элементу

`it1 == it2` и `it1 != it2`: операторы равенства и неравенства

`it--`: переходим к предыдущему элементу (работает не для всех итераторов)

`it + n`: только для итераторов `vector` – переходим к элементу, следующему через `n` элементов после текущего.

`it1 - it2`: только для итераторов `vector` – возвращает расстояние между элементами

Рассмотрим пример работы с итераторами. Приведённый ниже код создаёт массив (`vector`) и множество на основе бинарного дерева, заполняет их элементами и печатает. Если для вектора поведение итератора очень похоже на указатель, то для множества оно сильно отличается. Например, перегруженный оператор `++` для итератора `set` – это нетривиальная операция перехода к следующему элементу дерева.

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;

int main () {
    vector<int> v = {54, 62, 12, 97, 41, 6, 73};
    for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    set<int> s = {54, 62, 12, 97, 41, 6, 73};
    for (set<int>::iterator it = s.begin(); it != s.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
}
```

Задачи

- Напечатайте только чётные элементы вектора, используйте итераторы
- Напечатайте каждый второй элемент вектора, используйте итераторы
- Напишите функцию `inc`, которая будет принимать на вход вектор целых чисел типа `int` и будет увеличивать все элементы на 1. Для прохода по вектору используйте итераторы.
- Напечатать содержимое вектора в обратном порядке

`std::vector` и алгоритмы

В библиотеке `algorithm` содержится множество алгоритмов, предназначенных для работы с контейнерами STL.

`std::max_element` и `std::min_element`

Принимает на вход 2 итератора и возвращает итератор на максимальный элемент на подмассиве, задаваемом этими итераторами. Если таких элементов несколько, то возвращает итератор на первый из них.

Задачи:

- На вход подаётся n чисел. Напечатайте минимальный элемент и его индекс.

вход	выход
7 8 2 5 4 1 6 4	1 4
1 1	1 0

- На вход подаётся чётное количество чисел. Напечатайте минимальный элемент на первой половине и максимальный элемент второй половины.

вход	выход
8 7 2 8 4 1 9 4 2	2 9
8 8 7 6 5 4 3 2 1	5 4
2 5 1	5 1

- На вход подаётся n чисел. Напечатайте максимальный элемент, который находится до минимального. Предполагается, что минимальный элемент не является первым.

вход	выход
7 7 2 8 4 1 9 4	8
7 2 1 2 3 4 5 6	2
2 3 1	3

`std::find`

Принимает на вход 2 итератора и элемент того же типа, что и тип элементов вектора. Ищет этот элемент и возвращает итератор на этот элемент. Если этого итератора в контейнере нет, то возвращает итератор `end()` этого контейнера.

- На вход подаётся n чисел и ещё некоторое число. Напечатайте индекс этого числа в массиве. Если такого числа в массиве нет, то напечатайте `No such element`.

вход	выход
7 8 2 5 4 1 6 4 6	5
2 4 1 5	`No such element`

`std::sort`

Принимает на вход 2 итератора. Сортирует подмассив, задаваемый этими итераторами, по возрастанию. Сортировка работает за $O(n \log(n))$.

- На вход подаётся n чисел. Отсортируйте их и напечатайте.

вход	выход
8 8 2 5 4 9 1 6 4	1 2 4 4 5 6 8 9

- На вход подаётся n строк. Отсортируйте их лексиграфически и напечатайте.

вход	выход
5 Cat Dog Elephant Cattle Dolphin	Cat Cattle Dog Dolphin Elephant

`std::reverse`

Принимает на вход 2 итератора. Обращает подмассив, задаваемый этими итераторами.

- На вход подаётся n чисел. Найдите максимум и отсортируйте элементы, идущие до максимума по возрастанию, а все элементы, идущие после максимума – по убыванию.

вход	выход
8 8 2 5 4 9 1 6 4	2 4 5 8 9 6 4 1

`std::count`

Принимает на вход 2 итератора и некоторое значение. Находит сколько элементов массива равны этому значению.

- На вход подаётся n чисел. Найдите сколько элементов массива равны максимальному.

вход	выход
8 8 2 5 8 8 1 6 4	3

`std::accumulate` (библиотека `numeric`)

Принимает на вход 2 итератора и некоторый объект (начальное значение). Прибавляет все элементы из подмассива, задаваемого итераторами, к начальному значению. В итоге возвращает получившееся значение.

- На вход подаётся n чисел. Напечатайте сумму этих чисел.

вход	выход
8 8 2 5 4 9 1 6 4	39
3 2000000000 1000000000 2000000000	5000000000

- На вход подаётся n чисел и ещё целое число k. Напечатайте сумму k наименьших чисел.

вход	выход
8 8 2 5 4 9 1 6 4 3	7

std::pair

Пара – это простейший контейнер, который может хранить в себе 2 элемента (возможно, разных типов). Реализация пары имеет примерно следующий вид:

```
template <typename T1, typename T2> struct pair {
    T1 first;
    T2 second;
};
```

Для пары определены операторы сравнения. Сравнение происходит в лексиграфическом порядке. То есть для оператора больше сначала сравниваются первые элементы и только если они равны, сравниваются вторые.

Для простого создания пар есть шаблонная функция `make_pair`. Пару можно создать так:

```
std::pair<string, int> p1 = make_pair("Titanic", 8.4);
std::pair<string, int> p2 {"Titanic", 8.4};
```

- На вход подаётся n чисел. Отсортируйте их и напечатайте сами элементы и их старые индексы.

вход	выход
8	1 2 4 4 5 6 8 9
8 2 5 4 9 1 6 4	5 1 3 7 2 6 0 4

- На вход подаётся n фильмов. Передаются названия фильмов и их рейтинг на кинопоиске. Отсортировать эти фильмы по возрастанию рейтинга.

вход	выход
5	Venom2 6.2
TheMatrix 8.5	Shrek 8.0
Titanic 8.4	Titanic 8.4
GreenMile 8.9	TheMatrix 8.5
Shrek 8.0	GreenMile 8.9
Venom2 6.2	

std::list

`std::list` – Это двусвязный список. Основные методы для работы со списком:

метод	описание
`insert`	Принимает на вход итератор и некоторый объект и вставляет этот объект перед элементом, на который указывает итератор.
`erase`	Принимает на вход итератор и удаляет элемент. Переданный итератор, конечно, становится недействительным, поэтому этот метод возвращает корректный итератор на элемент, следующий за удалённым
`push_back`	Добавить элемент в конец.
`push_front`	Добавить элемент в начало.
`pop_back`	Удалить элемент из конца.
`pop_front`	Добавить элемент из начала.
`sort`	Сортирует список (функция `sort` из библиотеки `algorithm` для списка не работает)

К итераторам `std::list<T>::iterator` нельзя прибавлять целые числа, вместо этого нужно использовать функцию `std::advance`. Также эти итераторы нальзя вычитать, вместо этого нужно использовать функцию `std::distance`. Эти две функции работают за линейное время.

- На вход подаётся n чисел. Сохраните эти числа в связном списке, найдите их сумму и напечатайте её.
- На вход подаётся n чисел. Сохраните эти числа в связном списке, отсортируйте список и напечатайте их.
- На вход подаётся n чисел. Сохраните эти числа в связном списке. Скопируйте все элементы списка в его конец и напечатайте его.

вход	выход
5 8 2 1 4 2	8 2 1 4 2 8 2 1 4 2

- Напишите функцию `insert_after`, которая будет принимать на вход список из чисел типа `int`, итератор на элемент этого списка и некоторое число `x`. Функция доджна вставлять `x` после элемента, заданным итератором.
- Проверьте только что написанную функцию. На вход подаётся n чисел. Сохраните эти числа в связном списке. Продублируйте каждый элемент списка.

вход	выход
5 8 2 1 4 2	8 8 2 2 1 1 4 4 2 2

- На вход подаётся n чисел. Сохраните эти числа в связном списке. Удалите все чётные числа из списка и напечатайте его.

вход	выход
5 8 2 1 4 5	1 5