Семинар #4: Контейнеры STL.

Часть 1: Контейнеры

Стандартная библиотека шаблонов (STL = Standard Template Library) 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

Часть 2: 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

Часть 3: `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

Часть 4: `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 8 2 5 4 9 1 6 4	1 2 4 4 5 6 8 9 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 TheMatrix 8.5 Titanic 8.4 GreenMile 8.9 Shrek 8.0 Venom2 6.2	Venom2 6.2 Shrek 8.0 Titanic 8.4 TheMatrix 8.5 GreenMile 8.9

Часть 5: `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

Часть 6: `std::set`

`std::set` – это реализация множества с помощью бинарного дерева поиска. Не хранит дупликатов. При попытке добавить в множество тот элемент, который в нём уже есть, ничего не произойдёт. Также все элементы в множестве всегда хранятся в отсортированном виде (так как это бинарное дерево поиска). Для типа элементов множество должен быть реализован `operator<`. В `std::set` нельзя менять элементы, так как это бинарное дерево поиска, но можно удалить элемент, а потом вставить новый.

Основные методы для работы с множеством:

метод	описание
`insert`	Вставляет элемент в множество
`erase`	Удаляет элемент. Можно удалять по значению элемента или по итератору. Также можно сразу удалить диапазон значений, если передать 2 указателя
`find(x)`	Принимает на вход значение `x` и ищет такой элемент в множестве. Возвращает итератор на этот элемент или итератор `end()`, если такого элемента нет
`count(x)`	Принимает значение и находит, сколько элементов равны этому значению (т.е. 0 или 1)
`lower_bound(x)`	Возвращает итератор на первый элемент, который больше или равен `x`
`upper_bound(x)`	Возвращает итератор на первый элемент, который больше `x`

- На вход подаётся n чисел. Напечатайте эти числа удалив все дупликаты.

вход	выход
10 8 2 1 2 2 1 8 7 1 2	1 2 7 8

- На вход подаётся n чисел и некоторое число `x`. Найдите пару элементов массива, такую что их сумма равна `x`. Напечатайте индексы этих элементов. При наличии нескольких таких пар, напечатайте любую. Решение должно работать за $O(n \log(n))$ или быстрее.

вход	выход
8 8 2 5 4 9 1 7 4 14	2 4

`std::multiset`

То же самое, что и `std::set`, но может хранить дупликаты. Одна из неочевидных особенностей `multiset` это то, что при удалении элемента по значению `erase(x)`, удалятся все элементы, равные `x`. Для удаления одного элемента нужно передать в `erase` итератор на элемент.

- Считайте n чисел и отсортируйте их с помощью вставки в `multiset`. Распечатайте отсортированные числа.
- На прямой лежит верёвка длиной n метров. Затем её начинают последовательно разрезать. Все места разрезов – целые числа. Найти длину самого длинного куска после каждого разреза.

вход	выход
20 8 8 10 15 1 7 4 11 18	12 10 8 7 6 5 5 4

Часть 7: `std::map`

`std::map` – это реализация словаря с помощью бинарного дерева поиска. Не хранит ключе - дупликатов. При попытке добавить в этот словарь элемента с ключом, который в нём уже есть, ничего не произойдёт. Также все элементы в этом словаре всегда хранятся в отсортированном по ключам виде (так как это бинарное дерево поиска). Для типа ключей должен быть реализован `operator<`. В `std::map` можно менять значения, но нельзя менять ключи, так как это бинарное дерево поиска. Но можно удалить элемент каким-то ключом, а потом вставить новый с другим ключом.

Основные методы для работы с множеством:

метод	описание
`insert(k, v)`	Вставляет элемент с ключом `k` и значением `v` Если такой элемент уже есть, то ничего не делает
`operator[]` `m[k] = v`	Вставляет элемент с ключом `k` и значением `v` Если такой элемент уже есть, то меняет его значение
`erase(k)`	Удаляет элемент. Можно удалять по значению элемента или по итератору. Также можно сразу удалить диапазон значений, если передать 2 указателя
`find(x)`	Принимает на вход значение `x` и ищет элемент с таким ключом. Возвращает итератор на этот элемент или итератор `end()`, если такого ключа нет
`count(x)`	Принимает значение и находит, сколько ключей равны этому значению (т.е. 0 или 1)
`lower_bound(x)`	Возвращает итератор на первый элемент, который больше или равен `x`
`upper_bound(x)`	Возвращает итератор на первый элемент, который больше `x`

Пример программы, которая создаёт словарь из пар <название города, его население>. Строка выступает в качестве ключа, а целое число – в качестве значения.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main () {
    map<string, int> m = {{"London", 8900000}, {"Moscow", 12500000}, {"Milan", 4300000}};

    string cityName;
    while (true) {
        cin >> cityName;
        if (cityName == "q" || cityName == "quit") {
            break;
        }
        map<string, int>::iterator it = m.find(cityName);
        if (it == m.end()) {
            cout << "No such city" << endl;
        }
        else {
            cout << "City " << cityName << " population = " << it->second << endl;
        }
    }
}
```

- Напишите программу, которая будет в бесконечном цикле считывать слова и после каждого считывания печатать все уникальные слова, считанные ранее и количество таких слов. Например, если пользователь ввёл слово `Cat` три раза, слово `Dog` 1 раз и слово `Elephant` 2 раза. То после очередного считывания программа должна напечатать:

```
Dictionary:
Cat: 3
Dog: 1
Elephant: 2
```

- Считайте все слова из файла и напечатайте все уникальные слова и то, как часто они встречались в файле. Сохраните результат в новом файле. Для работы с файлами можно использовать функции `C`.

входной файл	выходной файл

`I'm having Spam, Spam, Spam, Spam, Spam, Spam,`
`Spam, baked beans, Spam, Spam, Spam and Spam.`
	`I'm 1`
	`Spam 1`
	`Spam, 9`
	`Spam. 1`
	`and 1`
	`beans, 1`
	`having 1`

Часть 8: Дополнительно об итераторах

Обратные итераторы

Проход по контейнеру в обратном порядке с использованием обычных итераторов может быть затруднителен. С обратными итераторами это сделать гораздо проще. Метод `rbegin` возвращает итератор на последний элемент. Метод `rend` возвращает итератор на фиктивный элемент, следующий до первого. Перегруженный оператор `++` перемещает итератор к предыдущему элементу.

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v {4, 8, 15, 16, 23, 42};

    for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); it++) {
        std::cout << *it << " ";
    }
}
```

Константные итераторы

Рассмотрим следующую ситуацию:

```
#include <iostream>
#include <vector>
int main() {
    const std::vector<int> v{ 4, 8, 15, 16, 23, 42 };

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        std::cout << *it << " ";
    }
}
```

Вектор `v` – константный, а его вызываемые методы `begin` и `end` константными не являются. Компилятор знает, что эти методы могут быть использованы для изменения вектора `v` и поэтому запрещает их использование. Решение – использование итератора `const_iterator` и методов `cbegin` и `cend`.

- Измените код выше чтобы он работал, поменяв обычный итератор на константный.
- Напишите функцию `printEven`, которая будет принимать вектор по константному указателю и печатать только чётные элементы.
- Напишите функцию `printOdd`, которая будет принимать множество по константному указателю и печатать только нечётные элементы. Можно ли в этом случае использовать обычные неконстантные итераторы.

Функция `std::copy`

Стандартная функция `std::copy` принимает на вход три итератора. Первые два итератора задают множество элементов которые нужно скопировать, а третий – место куда нужно их скопировать. Пример кода: (полная версия в файле `std_copy.cpp`).

```
std::vector<int> a { 4, 8, 15, 16, 23, 42 };
std::vector<int> b(a.size());
std::cout << b;

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

- Почему в конструктор вектора `b` передаётся `a.size()`? Что будет если его не передавать?
- Измените контейнер `a` с `vector` на `set` и скопируйте содержимое `a` в `b`.

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 { 4, 8, 15, 16, 23, 42 };
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.