

Семинар #5: Итераторы и контейнеры.

Итераторы

Контейнер в C++ – это объект, используемый для хранения других объектов и отвечающий за управление памятью, используемой содержащимися в нем объектами. Примерами контейнеров являются `std::vector<T>` или `std::array<T, Size>`.

Итератор – это один из часто используемых паттернов проектирования. Итератор представляет собой объект, используя который, мы можем получить доступ к элементам некоторого другого объекта.

Итераторы в C++ – это специальные объекты, которые используются для доступа к элементам контейнеров. Благодаря им мы можем, например, обойти все элементы контейнера или задать в этом контейнере некоторый диапазон. Для каждого контейнера есть свой тип итератора и этот тип определён внутри самого класса контейнера и называется `iterator`. То есть, если у нас есть контейнер `std::vector<int>`, то тип итератора для такого контейнера будет `std::vector<int>::iterator`. Также, у каждого контейнера есть специальные методы:

- `begin` – возвращает итератор на первый элемент
- `end` – возвращает итератор на фиктивный элемент, следующий за последним

Пример создания итератора вектора и работы с ним:

```
#include <iostream>
#include <vector>
using std::cout, std::endl;

int main()
{
    std::vector<int> v {10, 20, 30, 40, 50};
    std::vector<int>::iterator it = v.begin();
    cout << *it << endl; // Напечатает 10
    it++;
    cout << *it << endl; // Напечатает 20
}
```

Операции, которые можно проводить с итератором вектора

1. Копирование и присваивание: `Iterator it1 = it2;` `it1 = it2`
`it1` будет указывать туда же куда указывает `it2`.
2. Инкремент/декремент итератора: `++it` `it++` `--it` `it--`
В этом случае итератор начинает указывать на предыдущий или следующий элемент.
3. Прибавление/вычитание целого числа: `it += k` `it -= k`
В этом случае итератор начинает указывать на элемент, смещённый на это число.
4. Сумма/разность итератора и числа: `it + k` `it - k`
Результат этой операции – это новый итератор, который смещён на данное число.
5. Вычитание итераторов: `it1 - it2`
Возвращает количество элементов между этими объектами
6. Сравнения: `it1 == it2` `it1 != it2` `it1 < it2` ...
7. Унарная звёздочка: `*it`
Поставив `*` перед итератором мы получим объект, на который указывает итератор.
8. Оператор индексирования: `it[k]`
Также, как и для указателей, `it[k]` это то же самое, что и `*(it + k)`.

! Набор операций для других итераторов может сильно отличаться.

Пример прохода по вектору с использованием итератора

Конечно, можно пройти по вектору и с помощью обычной целочисленной переменной. Но можно сделать это и используя итераторы как показано в следующем примере.

```
#include <iostream>
#include <vector>
using std::cout, std::endl;

int main()
{
    std::vector<int> v {11, 22, 33, 44, 55};

    // Напечатаем все элементы вектора:
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        cout << *it << " ";
    cout << endl;

    // Увеличим все элементы вектора на 1:
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        *it += 1;

    // Напечатаем только чётные элементы:
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    {
        if (*it % 2 == 0)
            cout << *it << " ";
    }
    cout << endl;
}
```

Передача итераторов в функции

С итераторами можно работать как с обычными переменными. Их можно хранить отдельно от контейнера, можно передавать в функции, можно положить в другие контейнеры и т. д.

```
#include <iostream>
#include <vector>
using std::cout, std::endl;

void print(std::vector<int>::iterator first, std::vector<int>::iterator last)
{
    for (std::vector<int>::iterator it = first; it != last; ++it)
        cout << *it << " ";
    cout << endl;
}

int main()
{
    std::vector<int> v {11, 22, 33, 44, 55};
    print(v.begin(), v.end());           // Напечатает весь вектор
    print(v.begin(), v.begin() + 3);     // Напечатает первые 3 элемента
}
```

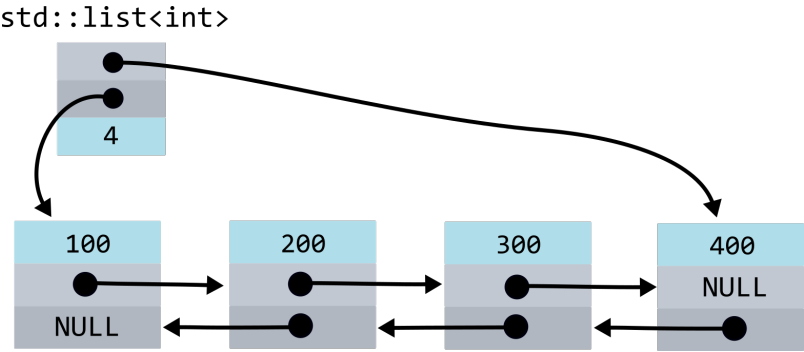
Контейнеры

Стандартная библиотека включает в себя множество разных шаблонных контейнеров.

контейнер	описание и основные свойства
<code>std::vector</code>	Динамический массив, по умолчанию хранит элементы в куче. Все элементы лежат вплотную друг к другу. Доступ по индексу за $O(1)$. Вставка/удаление в конец за $O(1)$ в среднем. В остальных случаях вставка/удаление за $O(n)$. Поиск за $O(n)$.
<code>std::array</code>	Массив фиксированного размера, хранит элементы в самом объекте. Все элементы лежат вплотную друг к другу. Доступ по индексу за $O(1)$. Поиск за $O(n)$.
<code>std::list</code>	Двусвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на элемент. Нет доступа по индексу. Поиск за $O(n)$.
<code>std::forward_list</code>	Односвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на предыдущий элемент. Нет доступа по индексу. Поиск за $O(n)$.
<code>std::deque</code>	Двухсторонняя очередь Доступ по индексу за $O(1)$. Добавление/удаление в начало и конец за $O(1)$. Остальные операции за $O(N)$.
<code>std::set</code>	Реализация множества на основе сбалансированного дерева поиска. Хранит элементы без дубликатов, в отсортированном виде. Тип элементов должен реализовать <code>operator<</code> (или предоставить компаратор). Поиск/вставка/удаление элементов за $O(\log(N))$.
<code>std::map</code>	Реализация словаря на основе сбалансированного дерева поиска. Хранит пары ключ-значения без дубликатов ключей, в отсортированном виде. Тип ключей должен реализовать <code>operator<</code> (или предоставить компаратор). Поиск/вставка/удаление элементов за $O(\log(N))$.
<code>std::unordered_set</code>	Реализация множества на основе хеш-таблицы. Хранит элементы без дубликатов, в произвольном порядке. Поиск/вставка/удаление элементов за $O(1)$ в среднем.
<code>std::unordered_map</code>	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.
<code>std::multiset</code> <code>std::multimap</code>	То же самое, что <code>std::set</code> / <code>std::map</code> , но может хранить дублированные значения

Контейнер `std::list`

Контейнер `std::list` реализует двусвязный список. Его строение можно представлять следующим образом:



Основные методы для работы со списком. Все перечисленные методы работают за $O(1)$.

метод	описание
<code>size_t size()</code>	возвращает количество элементов в списке. работает за $O(1)$, так как количество элементов хранится внутри списка.
<code>void push_back(const T& el)</code>	добавляет элемент в конец списка.
<code>void pop_back()</code>	удаляет элемент из конца списка.
<code>void push_front(const T& el)</code>	добавляет элемент в начало списка.
<code>void pop_front()</code>	удаляет элемент из начала списка.
<code>iterator insert(iterator it, const T& elem)</code>	вставляет элемент до элемента на который указывает итератор <code>it</code> . Возвращает итератор на новый элемент.
<code>iterator erase(iterator it)</code>	удаляет элемент на который указывает <code>it</code> . возвращает итератор, указывающий на следующий за удалённым.

Преимущество списка по сравнению с массивом в том, что можно быстро добавлять и удалять элементы в любое место списка (если это положение известно), тогда как в массив можно быстро добавлять/удалять только в конец.

Обратите внимание, что у списка нет оператора индексации `operator[]`, так как в связном списке нет быстрого способа получить доступ к элементу по его индексу. Если бы такой метод был бы написан, то он работал бы за $O(n)$, что очень плохо. Поэтому для обхода связного списка нужно использовать итераторы:

```
#include <iostream>
#include <list>
int main()
{
    std::list<int> a {10, 20, 30, 40};
    a.push_back(50);
    // Напечатаем все элементы списка: 10 20 30 40 50
    for (std::list<int>::iterator it = a.begin(); it != a.end(); ++it)
        std::cout << *it << " ";
    std::cout << std::endl;
}
```

Пример работы со связным списком

Пример использования методов, описанных выше:

```
#include <iostream>
#include <list>
int main()
{
    std::list<int> a {99, 20, 30, 40, 50};

    a.pop_front();
    a.push_front(10);

    std::list<int>::iterator it = a.begin();
    it++;
    it++;
    a.insert(it, 99);

    // Напечатает 10 20 99 30 40 50
    for (std::list<int>::iterator it = a.begin(); it != a.end(); ++it)
        std::cout << *it << " ";
    std::cout << std::endl;
}
```

Обход связного списка с удалением/добавлением элементов

Сложность при использовании связного списка и других контейнеров может возникнуть если вы обходите список и при этом его меняете. Например, рассмотрим задачу удаления всех чётных элементов из списка.

```
#include <iostream>
#include <list>
int main()
{
    std::list<int> a {11, 22, 33, 44, 55};

    // Неправильный способ. Потому что после удаления элемента итератор на него
    // стал недействительным. Применение ++it к такому итератору приводит к UB.
    for (std::list<int>::iterator it = a.begin(); it != a.end(); ++it)
    {
        if (*it % 2 == 0)
            a.erase(it);
    }

    // Правильный способ.
    for (std::list<int>::iterator it = a.begin(); it != a.end(); )
    {
        if (*it % 2 == 0)
            it = a.erase(it);
        else
            it++;
    }

    for (std::list<int>::iterator it = a.begin(); it != a.end(); ++it)
        std::cout << *it << " ";
    std::cout << std::endl;
}
```

Операции, которые можно проводить с итератором списка

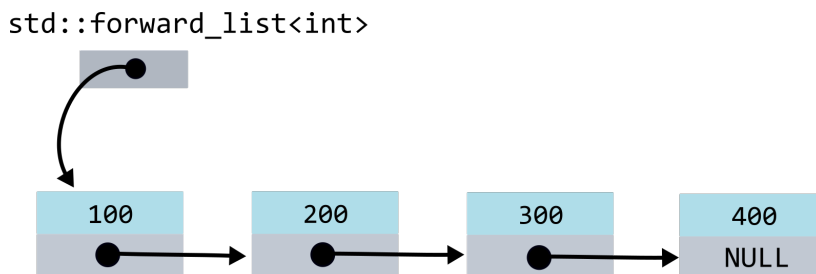
Набор операций итератора списка значительно меньше, чем у итератора вектора:

1. Копирование и присваивание: `Iterator it1 = it2; it1 = it2`
`it1` будет указывать туда же куда указывает `it2`.
2. Инкремент/декремент итератора: `++it it++ --it it--`
В этом случае итератор начинает указывать на предыдущий или следующий элемент.
3. Сравнения на равенство/неравенство: `it1 == it2 it1 != it2`
Но нельзя сравнивать на больше/меньше.
4. Унарная звёздочка: `*it`
Поставив `*` перед итератором мы получим объект, на который указывает итератор.

К итератору списка нельзя прибавлять числа, нельзя вычитать итераторы, сравнивать и применять оператор индексирования. Легко понять почему эти операции для итератора списка не реализованы, если знать внутреннее устройство связанного списка – их нельзя реализовать эффективно. Например, операция `it + n` даже если бы она была реализована, работала бы за $O(n)$, так как, чтобы сместиться вперёд на n элементов в списке, нужно сделать n переходов по указателю.

Контейнер `std::forward_list`

Контейнер `std::forward_list` реализует односвязный список. Его строение можно представлять следующим образом:



Это более легковесный контейнер по сравнению `std::list`, но менее функциональный. Например, у него нет методов `size`, `push_back` и `pop_back`, а методы `insert` и `erase` заменены на `insert_after` и `erase_after`.

Операции, которые можно проводить с итератором односвязного списка

Набор операций, которые можно проводить с итератором односвязного списка ещё меньше чем у двусвязного:

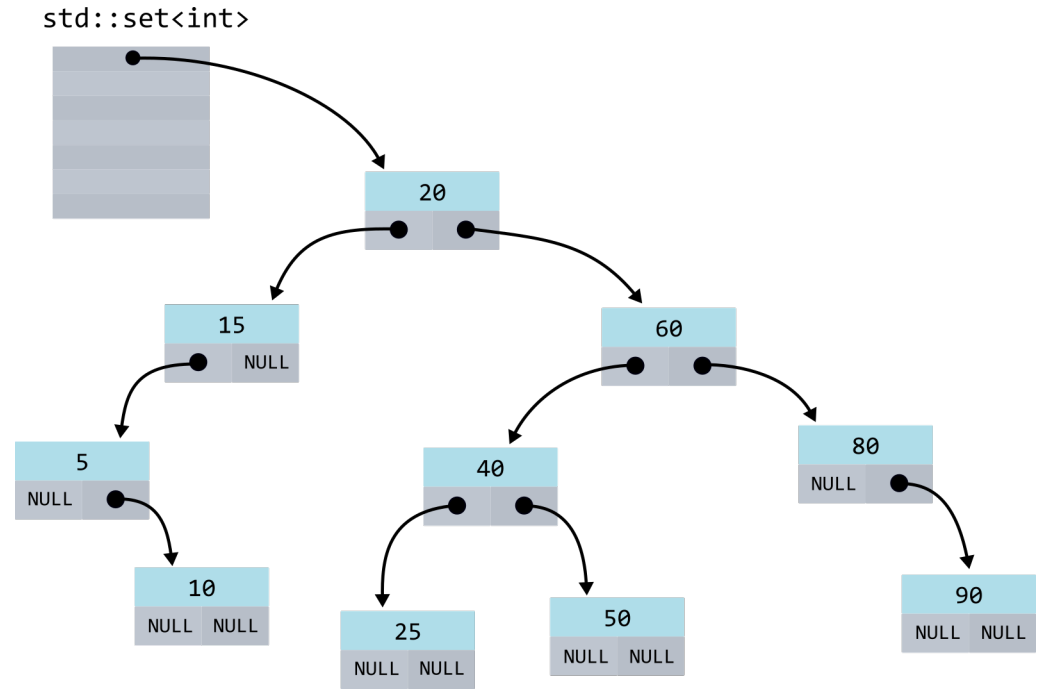
1. Копирование и присваивание: `Iterator it1 = it2; it1 = it2`
2. Инкремент итератора: `++it it++`
3. Сравнения на равенство/неравенство: `it1 == it2 it1 != it2`
Но нельзя сравнивать на больше/меньше.
4. Унарная звёздочка: `*it`

В отличие от итератора двусвязного списка к итератору односвязного списка нельзя применять декремент (`--`).

```
#include <iostream>
#include <forward_list>
int main()
{
    std::forward_list<int> a {10, 20, 30, 40, 50};
    std::forward_list<int>::iterator it = a.begin();
    it++;
    it--; // Ошибка компиляции
}
```

Контейнер `std::set`

`std::set` — это реализация множества с помощью бинарного дерева поиска. Не хранит дубликатов. 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. Все следующие операции работают за $O(\log(n))$.

метод	описание
<code>std::pair<iterator, bool> insert(const T& el)</code>	Вставляет элемент в множество. Возвращает пару (итератор, булево значение). Итератор будет указывать на соответствующий элемент. Булево значение будет равно <code>true</code> если вставка была произведена и <code>false</code> если элемент уже существовал.
<code>iterator erase(iterator it)</code> <code>iterator erase(const T& el)</code>	Удаляет элемент. Можно удалять по итератору или по значению элемента.
<code>iterator find(const T& el)</code>	Ищет элемент в множестве. Возвращает итератор на этот элемент или итератор <code>end()</code> , если такого элемента нет.
<code>bool contains(const T& el)</code>	Возвращает <code>true</code> , если элемент содержится в множестве.
<code>iterator lower_bound(x)</code>	Возвращает итератор на первый элемент, который больше или равен <code>x</code>
<code>iterator upper_bound(x)</code>	Возвращает итератор на первый элемент, который больше <code>x</code>

Преимущество множества над вектором заключается в том, что операции вставки/удаления/поиска работают за $O(\log(n))$, что намного быстрее чем у вектора (за исключения вставки/удаления в конец вектора). Недостатком множества является то, что в нём нельзя быстро найти элемент по индексу.

Пример работы с множеством

```
#include <iostream>
#include <set>

void print_set(std::set<int>::iterator start, std::set<int>::iterator finish)
{
    for (std::set<int>::iterator it = start; it != finish; ++it)
        std::cout << *it << " ";
    std::cout << std::endl;
}

int main()
{
    std::set<int> a {40, 20, 10, 10, 50, 30, 30};
    print_set(a.begin(), a.end()); // Напечатает 10 20 30 40 50

    a.insert(20);
    a.insert(60);
    a.erase(10);
    print_set(a.begin(), a.end()); // Напечатает 20 30 40 50 60

    std::set<int>::iterator it = a.find(50);
    if (it == a.end())
    {
        std::cout << "Element not found" << std::endl;
    }
    else
    {
        std::cout << "Element found. Printing elements starting with this one:" << std::endl;
        print_set(it, a.end());
    }
}
```

Контейнер std::multiset

То же самое, что и `std::set`, но может хранить дубликаты. 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.

```
#include <iostream>
#include <set>

int main()
{
    std::multiset<int> a {40, 20, 10, 10, 50, 30, 30};
    // В мультимножестве будет лежать 10 10 20 30 30 40 50

    a.erase(10);          // Удалит все элементы, равные 10
    a.erase(a.find(30));  // Удалит один элемент, равный 30

    // В мультимножестве будет лежать 20 30 40 50
}
```


Контейнер `std::map`

`std::map` – это реализация словаря с помощью бинарного дерева поиска. Не хранит ключей - дубликатов. 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 со словарём:

метод	описание
<code>m[key]</code>	Получить значение по ключу
<code>m[key] = newValue</code>	Изменить значение с ключом <code>key</code> . Если элемента с таким ключом в словаре нет, то новый элемент с таким ключом вставится в словарь.
<code>std::pair<iterator, bool> insert(const T& el)</code>	Вставляет элемент в словарь. Тут <code>T</code> – это <code>std::pair<Key, Value></code> . Если элемент уже существовал, то ничего не делает. Возвращает пару (итератор, булево значение). Итератор будет указывать на соответствующий элемент. Булево значение будет равно <code>true</code> если вставка была произведена и <code>false</code> если элемент уже существовал.
<code>iterator erase(iterator it)</code> <code>iterator erase(const K& key)</code>	Удаляет элемент. Можно удалять по значению элемента или по итератору.
<code>iterator find(const K& key)</code>	Ищет элемент с таким ключом <code>key</code> . Возвращает итератор на этот элемент или итератор <code>end()</code> , если элемента с таким ключом нет.
<code>bool contains(const K& key)</code>	Проверяет, существует ли элемент с ключом <code>key</code> в словаре.
<code>iterator lower_bound(x)</code>	Возвращает итератор на первый элемент, который больше или равен <code>x</code> .
<code>iterator upper_bound(x)</code>	Возвращает итератор на первый элемент, который больше <code>x</code> .

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 std::cout, std::endl;

int main ()
{
    std::map<string, int> m = {{"London", 9}, {"Moscow", 12}, {"Milan", 4}};

    std::string cityName;
    while (true)
    {
        std::cin >> cityName;
        if (cityName == "q" || cityName == "quit")
            break;

        std::map<std::string, int>::iterator it = m.find(cityName);
        if (it == m.end())
            cout << "No such city" << endl;
        else
```

```
        cout << "City " << cityName << " population = " << it->second << endl;
    }
}
```

- 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	2 4
8 2 5 4 9 1 7 4	
14	

- 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.

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

Ключевое слово auto

Ключевое слово `auto` используется для автоматического вывода типа.

```
#include <string>
int main()
{
    auto a = 123;    // a будет иметь тип int
    auto b = 4.1;    // b будет иметь тип double
    auto b = 4.1f;   // b будет иметь тип float

    auto s1 = "Hello";           // s1 будет иметь тип const char*
    auto s2 = std::string("Hello"); // s2 будет иметь тип std::string
}
```

Задачи:

- В примере ниже создан вектор строк и напечатано его содержимое. Тип итератора имеет очень длинное название (и название будет ещё больше если контейнер будет хранить не просто строки, а что-нибудь посложнее). Используйте `auto`, чтобы упростить код.

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

int main()
{
    std::vector<std::string> v {"Cat", "Dog", "Elephant"};
    for (std::vector<std::string>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << *it << std::endl;
}
```

- Протестируйте, можно ли использовать `auto` вместо возвращаемого типа функции. Напишите функцию, которая принимает на вход вектор строк и возвращает строку, которая является результатом конкатенации всех строк. Вместо возвращаемого типа используйте `auto`.
- Протестируйте, можно ли создать функцию, которая будет принимать целое число и, в зависимости от этого числа, возвращать значения разных типов. (Если вместо возвращаемого типа используется `auto`).
- Протестируйте, можно ли использовать `auto` для указателя с помощью `auto`. Пусть есть такой участок кода:

```
int a = 123;
auto p = &a;
auto* q = &a;
```

Какой тип будет у `p` и `q`?

- Функция вычисления факториала, написанная ниже с использованием `auto` не работает.

```
auto factorial(int n)
{
    if (n > 0)
        return n * factorial(n - 1);
    return 1;
}
```

Почему? Исправьте эту функцию, не убирая `auto`.

Range-based циклы

Циклы, основанные на диапазоне, предоставляют более простой способ обхода контейнера:

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

int main()
{
    std::vector v {6, 1, 7, 4};
    for (int num : v)
        std::cout << num << std::endl;
}
```

Для изменения элементов контейнера при обходе нужно использовать ссылки:

```
for (int& num : v)
    num += 1;
```

Задачи:

- Проверьте, можно ли использовать ключевое слово `auto` внутри таких циклов.
- Пусть у нас есть вектор строк:

```
vector<string> v {"Cat", "Axolotl", "Bear", "Elephant"};
```

- Напишите range-based цикл, который будет печатать все элементы вектора
- Напишите range-based цикл, который будет добавлять в конец каждой строки символ `s`.
- Напишите range-based цикл, который будет обращать каждую строку. Используйте стандартную функцию `reverse`.

- Проверьте, можно ли использовать range-based циклы если контейнер является:

- | | |
|--------------------------|----------------------------|
| – <code>std::list</code> | – Обычным массивом |
| – <code>std::set</code> | – <code>std::string</code> |
| – <code>std::map</code> | |
| – <code>std::pair</code> | – Строкой в стиле C |

- Для печати массива целых чисел была написана следующая функция:

```
void print(int array[])
{
    for (int num : array)
        std::cout << num << std::endl;
}
```

Оказывается, что она не работает. В чём заключается ошибка?

Structure binding (структурное связывание)

В стандарте C++17 был добавлен новый вид объявления и инициализации нескольких переменных. В коде ниже мы объявляем переменные `a` и `b` одной строкой с помощью структурного связывания.

```
#include <iostream>
#include <utility>

int main()
{
    std::pair p {5, 1};
    auto [a, b] = p;

    std::cout << a << " " << b << std::endl;
}
```

Структурное связывание работает только в том случае, если размер контейнера справа известен на стадии компиляции. Например, пары, кортежи(`std::tuple`), статические массивы, `std::array`, простые структуры.

Задачи:

- Пусть у нас есть пара:

```
std::pair p {std::string{"Moscow"}, 1147};
```

- Создайте две переменные `name` и `age` и присвойте их соответствующим элементам пары.
- Создайте две ссылки `name` и `age` и инициализируйте их соответствующими элементами пары. Убедитесь, что при изменении переменной `name` меняется и пара `p`.
- Метод `insert` контейнера `std::set` пытается вставить элемент в множество. Если же такой элемент в множестве уже существует, то он ничего с множеством не делает. Но этот метод возвращает пару из итератора на соответствующий элемент и переменной типа `bool`, которая устанавливается в `true` если новый элемент был добавлен и в `false`, если такой элемент уже существовал. Вот пример программы, которая пытается вставить элемент в множество и печатает соответствующее сообщение. В любом случае программа печатает все элементы, меньшие вставляемого.

```
#include <iostream>
#include <utility>
#include <set>
using std::cout, std::endl;

int main()
{
    std::set<int> s {1, 2, 4, 5, 9};

    std::pair<std::set<int>::iterator, bool> result = s.insert(5);
    if (result.second == true)
        cout << "Element added successfully" << endl;
    else
        cout << "Element already existed" << endl;

    for (std::set<int>::iterator it = s.begin(); it != result.first; ++it)
        cout << *it << " ";
}
```

Упростите эту программу, используя ключевое слово `auto` и структурное связывание.

Структурное связывание можно использовать и в цикле.

```
#include <iostream>
#include <utility>
#include <vector>

int main()
{
    std::vector<std::pair<std::string, int>> v {"Moscow", 1147}, {"Berlin", 1237},
                                              {"Rome", -753}, {"Bogota ", 1538}};

    for (auto [city, year] : v)
        std::cout << city << " " << year << std::endl;
}
```

Задачи:

- В файле `books.cpp` лежит заготовка кода. В ней содержится инициализированный массив из структур. Сделайте следующее:
 - Напечатайте массив `books`, используя range-based цикл. Нужно напечатать все поля через запятую.
 - Напечатайте массив `books`, используя range-based цикл со структурным связыванием.
 - Увеличьте поле `price` всех книг на одну величину, используя range-based цикл.
 - Увеличьте поле `price` всех книг на одну величину, используя range-based цикл со структурным связыванием.
- Ниже есть пример программы – решение задачи с предыдущего семинара. Она считывает слова и печатает количества всех введённых до этого слов. Упростите код этой программы, используя `auto` и структурное связывание.

```
#include <iostream>
#include <map>
#include <utility>
#include <string>
using std::cout, std::endl;

int main()
{
    std::map<std::string, int> wordCount;
    while (true)
    {
        std::string word;
        std::cin >> word;
        std::pair<std::string, int> wc {word, 1};
        std::pair<std::map<std::string, int>::iterator, bool> p = wordCount.insert(wc);
        if (p.second == false)
            wordCount[word] += 1;

        cout << "Dictionary:" << endl;
        for (std::map<std::string, int>::iterator it = wordCount.begin();
             it != wordCount.end(); ++it)
        {
            cout << (*it).first << ": " << (*it).second << endl;
        }
        cout << endl;
    }
}
```