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>

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

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 {10, 20, 30, 40, 50};
    for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); it++)
        std::cout << *it << " ";
}
```

Константные итераторы

Пусть есть вектор, который является константным. На элементы такого вектора нельзя указывать обычным итератором. `std::vector<T>::iterator`.

Также, нельзя использовать постоянный итератор `const std::vector<T>::iterator`. Ведь такой итератор будет сам постоянным, но с помощью него можно будет менять элементы вектора.

Чтобы работать с константным вектором необходимо использовать так называемые константные итераторы `std::vector<T>::const_iterator`. Такие итераторы могут меняться сами, но не могут менять вектор.

```
#include <iostream>
#include <vector>
int main()
{
    const std::vector<int> v {10, 20, 30, 40, 50};
    std::vector<int>::iterator it = v.begin();        // Ошибка, нельзя указывать обычным
                                                      // итератором на константный вектор

    const std::vector<int>::iterator it = v.begin();  // Ошибка, const делает сам итератор
                                                      // константным, но он всё ещё может
                                                      // менять вектор

    std::vector<int>::const_iterator it = v.begin();  // OK, const_iterator может меняться
                                                      // сам, но не может менять вектор

    auto it = v.begin();                              // OK, так как v константный, то метод
                                                      // begin возвращает const_iterator

    auto it = v.cbegin();                             // OK, метод cbegin будет возвращать
                                                      // const_iterator независимо от того
                                                      // константен вектор или нет

    // Обход вектора с помощью константного итератора
    for (std::vector<int>::const_iterator it = v.begin(); it != v.end(); it++)
        std::cout << *it << " ";
}
```

Аналогично, существует `const_reverse_iterator` – константный обратный итератор.

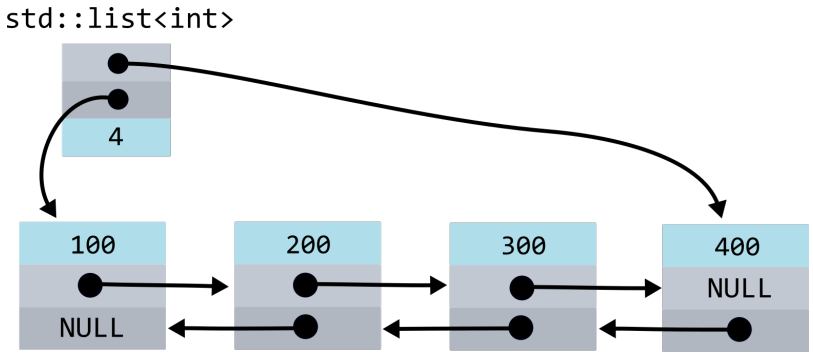
Контейнеры

Стандартная библиотека включает в себя множество разных шаблонных контейнеров.

контейнер	описание и основные свойства
`std::vector`	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 $O(n)$.
`std::array`	Массив фиксированного размера, хранит элементы в самом объекте. Все элементы лежат вплотную друг к другу. Доступ по индексу за $O(1)$. Поиск за $O(n)$.
`std::list`	Двусвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на элемент. Нет доступа по индексу. Поиск за $O(n)$.
`std::forward_list`	Односвязный список Вставка/удаление элементов за $O(1)$ если есть итератор на предыдущий элемент. Нет доступа по индексу. Поиск за $O(n)$.
`std::deque`	Двухсторонняя очередь Доступ по индексу за $O(1)$. Добавление/удаление в начало и конец за $O(1)$. Остальные операции за $O(N)$.
`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` `std::multimap`	То же самое, что `std::set`/`std::map`, 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
`size_t size()`	возвращает количество элементов в списке. работает за $O(1)$, так как количество элементов хранится внутри списка.
`void push_back(const T& el)`	добавляет элемент в конец списка.
`void pop_back()`	удаляет элемент из конца списка.
`void push_front(const T& el)`	добавляет элемент в начало списка.
`void pop_front()`	удаляет элемент из начала списка.
`iterator insert(iterator it, const T& elem)`	вставляет элемент до элемента на который указывает итератор `it`. Возвращает итератор на новый элемент.
`iterator erase(iterator it)`	удаляет элемент на который указывает `it`. 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

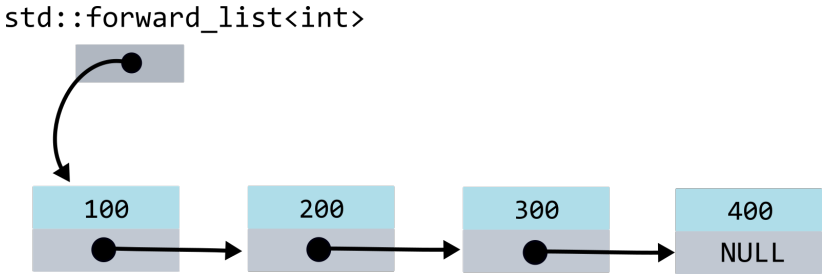
Набор операций итератора списка значительно меньше, чем у итератора вектора:

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 внутренее устройство связного 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:

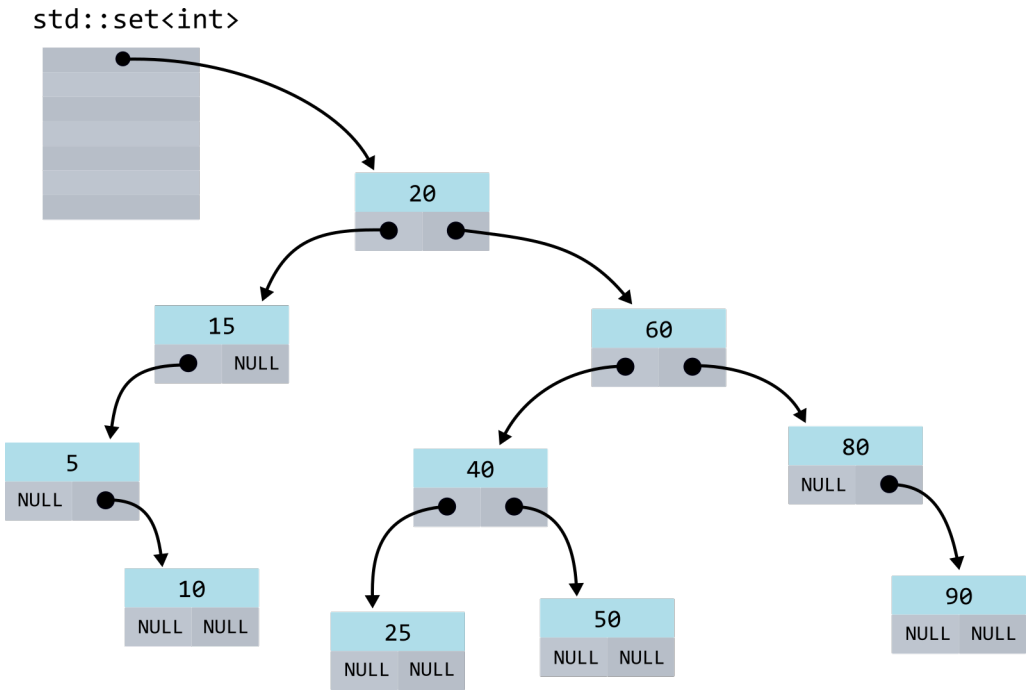
1. Копирование и присваивание: `Iterator it1 = it2;` `it1 = it2`
2. Инкремент итератора: `++it` `it++`
3. Сравнения на равенство/неравенство: `it1 == it2` `it1 != it2`
   Но нельзя сравнивать на больше/меньше.
4. Унарная звёздочка: `*it`

В отличии 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 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. Все следующие операции работают за $O(log(n))$.

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

Преемущество 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 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.

```
#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 - 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 со словарём:

метод	описание
`m[key]`	Получить значение по ключу
`m[key] = newValue`	Изменить значение с ключом `key`. Если элемента с таким ключом в словаре нет, то новый элемент с таким ключом вставится в словарь.
`std::pair<iterator, bool> insert(const T& el)`	Вставляет элемент в словарь. Тут `T` – это `std::pair<Key, Value>`. Если элемент уже существовал, то ничего не делает. Возвращает пару (итератор, булевое значение). Итератор будет указывать на соответствующий элемент. Булевое значение будет равно `true` если вставка была произведена и `false` если элемент уже существовал.
`iterator erase(iterator it)` `iterator erase(const K& key)`	Удаляет элемент. Можно удалять по значению элемента или по итератору.
`iterator find(const K& key)`	Ищет элемент с таким ключом `key`. Возвращает итератор на этот элемент или итератор `end()`, если элемента с таким ключом нет.
`bool contains(const K& key)`	Проверяет, существует ли элемент с ключом `key` в словаре.
`iterator lower_bound(x)`	Возвращает итератор на первый элемент, который больше или равен `x`.
`iterator 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 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 8 2 5 4 9 1 7 4 14	2 4

- 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

Определение контейнера

Внутренние типы контейнеров

- `iterator`
- `const_iterator`
- `value_type`
- `size_type`
- `difference_type`
- `reference`
- `const_reference`

Необходимые методы контейнера

Написание обобщённого кода с использованием контейнеров