Итераторы

Итератор — это объект, который способен перебирать элементы контейнерного класса без необходимости пользователю знать реализацию определенного контейнерного класса. Во многих контейнерах (особенно в списке и в ассоциативных контейнерах) итераторы являются основным способом доступа к элементам этих контейнеров. Об итераторе можно думать, как об указателе на определенный элемент контейнерного класса с дополнительным набором перегруженных операторов для выполнения определенных функций.

У итератора обычно следующий набор функций:

Перегружен оператор `*` . Он возвращает элемент, на который в данный момент указывает итератор. Оператор `++` перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют оператор `--` для перехода к предыдущему элементу. Также возможно использовать `std::next(iter, n)`, что эквивалентно `iter = iter + n` и `std::prev(iter, n)`, что эквивалентно `iter = iter - n`.

Операторы `==` и `!=` используются для определения того, указывают ли оба итератора на один и тот же элемент или нет. Для сравнения значений, на которые указывают оба итератора, нужно сначала разыменовать эти итераторы с помощью *, а затем использовать оператор `==` или оператор `!=`.

Оператор `=` присваивает итератору новую позицию (обычно начальный или конечный элемент контейнера). Чтобы присвоить значение элемента, на который указывает итератор, другому объекту, нужно сначала разыменовать итератор, а затем использовать оператор `=`.

Каждый контейнерный класс имеет 4 основных метода для работы с оператором =:

метод `begin()` возвращает итератор, представляющий начальный элемент контейнера;

метод `end()` возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере;

метод `cbegin()` возвращает константный (только для чтения) итератор, представляющий начальный элемент контейнера;

метод `cend()` возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

Вызов `end()` возвращает итератор, указывающий на элемент контейнера за последним. Это упрощает использование итераторов в цикле, итерации могут идти до тех пор, пока итератор не достигнет результата `end()`.

Существуют различные категории итераторов, такие как `RandomAccessIterator` или `ForwardIterator`.
Категория итератора определяет какие операции поддерживаются итератором:

- `RandomAccessIterator` умеет `it + n`, `it += n`, `++it`, `it--`, и т.п.;
- `BidirectionalIterator` может перемещаться только на один элемент: `++it`, `it--`;
- `ForwardIterator` может перемещаться только вперед: `++it` или `it++`.

Функции `std::advance`, `std::next` и `std::prev` упрощают перемещение между несколькими элементами для BidirectionalIterator и более простых категорий итераторов.

Все контейнеры предоставляют два типа итераторов:

`container::iterator` — итератор для чтения/записи;
`container::const_iterator` — итератор только для чтения.

Пример:

```
std::vector<int> vec;

for (int count=0; count < 5; ++count)
    vec.push_back(count);

std::vector<int>::const_iterator it;
it = vec.begin();// начальный элемент вектора
while (it != vec.end()) // пока элементы есть
{
    std::cout << *it << std::endl; // вывод элемента
    ++it; // переход к следующему элементу
}
```

В случае контейнера `std::map` в качестве элемента используются пары ключ-значение, задаваемые с помощью `std::pair`. Например:

```
std::map<int, std::string> myMap;
myMap.insert(std::make_pair(3, "cat"));
myMap.insert(std::make_pair(2, "dog"));
myMap.insert(std::make_pair(5, "chicken"));
myMap.insert(std::make_pair(4, "lion"));
myMap.insert(std::make_pair(1, "spider"));
```

```
std::map<int, std::string>::const_iterator it; // объявляем итератор
it = myMap.begin(); // присвоение начального элемента вектора
while (it != myMap.end())// пока не достигнет последнего элемента
{
   std::cout << it->first << "=" << it->second << std::endl; // вывод значение элемента, на который указывает итератор
   ++it; // переход к следующему элементу
}
```

Стандартные контейнеры, std::array

Представленный в C++11, `std::array` — это фиксированный массив. `std::array` определяется в заголовочном файле `array`, внутри пространства имен `std`.

```
std::array<int, 4> arr; // массив типа int длиной 4
std::array<int, 4> arr = { 8, 6, 4, 1 }; // список инициализаторов
std::array<int, 4> arr { 8, 6, 4, 1 }; // uniform-инициализация
std::array<int, > arr = { 8, 6, 4, 1 }; // ошибка, должна быть указана длина массива
```

Также можно присваивать значения массиву с помощью списка инициализаторов:

```
std::array<int, 4> arr;
arr = { 0, 1, 2, 3 }; // ок
arr = { 8, 6 }; // элементам 2 и 3 присвоен ноль
arr = { 0, 1, 3, 5, 7, 9 }; // ошибка, слишком много элементов в списке инициализаторов
```

Доступ к значениям массива может осуществляется через оператор индекса:

```
cout << arr[1] << endl;
arr[2] = 7;
```

В стандартных фиксированных массивах, оператор индекса не выполняет никаких проверок на диапазон. Если указан недопустимый индекс, то произойдёт ошибка.

`std::array` поддерживает вторую форму доступа к элементам массива — функция `at()`, которая осуществляет проверку диапазона:

```
std::array<int, 4> arr { 8, 6, 4, 1 };
arr.at(1) = 7;
arr.at(8) = 15; // элемент массива под номером 8 - некорректный, ошибка
```

Вызов `arr.at(1)` проверяет, есть ли элемент массива под номером `1`, и возвращается ссылка на этот элемент затем присваивается значение `7`. Однако, вызов `arr.at(8)` не срабатывает, так как элемента под номером `8` в массиве нет. Вместо возвращения ссылки, функция `at()` вернёт ошибку, которая завершает работу программы (выбрасывается исключение типа `std::out_of_range`).

Поскольку длина массива всегда известна, то циклы `foreach` также можно использовать с `std::array`:

```
std::array<int, 4> arr { 8, 6, 4, 1 };

for ( auto &element : arr ){
    cout << element << endl;
}
```

Массив можно отсортировать используя функцию `std::sort()`, которая находится в заголовочном файле `algorithm`:

```
std::array<int, 5> arr { 8, 4, 2, 7, 1 };
std::sort(arr.begin(), arr.end()); // сортировка массива по возрастанию
std::sort(arr.rbegin(), arr.rend()); // сортировка массива по убыванию
```

std::vector

Представленный в C++03, `std::vector` — это динамический массив, который может сам управлять выделенной себе памятью. Возможно создавать массивы, длина которых задается во время выполнения, без использования операторов `new` и `delete` (явного указания выделения и освобождения памяти). `std::vector` находится в заголовочном файле `vector`.

```
std::vector<int> array;
std::vector<int> array2 = { 10, 8, 6, 4, 2 };
std::vector<int> array3 { 10, 8, 6, 4, 2, 1 };
```

Подобно `std::array`, доступ к элементам массива может выполняться как через оператор `[ ]` (который не выполняет проверку диапазона), так и через функцию `at()` (которая выполняет проверку диапазона):

```
array[7] = 3; // без проверки диапазона
array.at(8) = 4; // с проверкой диапазона
```

Начиная с C++11, стало возможно присваивать значения для `std::vector`, используя список инициализаторов:

```
array = { 0, 2, 4, 5, 7 }; // длина array теперь 5
array = { 11, 9, 5 }; // длина array теперь 3
```

Длина в `std::vector` — это количество фактически используемых элементов, также ёмкость — это количество выделенных элементов.

```
vector<int> array { 12, 10, 8, 6, 4, 2 };
cout << "length is: " << array.size() <<
endl; // 6
cout << "capacity is: " << array.capacity() <<
endl; // 6
```

Изменить длину `std::vector` возможно с помощью функции `resize()`:

```
vector<int> array { 0, 1, 2 }; // 0,1,2
array.resize(7); // 0,1,2,0,0,0,0
array.resize(2); // 0,1
```

Поскольку изменение размера вектора является затратной операцией, то мы можем сообщить вектору выделить заранее заданный объем ёмкости, используя функцию `reserve()`:

```
vector<int> arr;
arr.reserve(7);  // ёмкость равна 7
```

При изменении вектором своего размера, он может выделить больше ёмкости, чем требуется. Это делается для обеспечения резерва для дополнительных элементов, чтобы свести к минимуму количество операций изменения размера.

```
vector<int> arr = { 0, 1, 2, 3, 4, 5 };
cout << "size: " << arr.size() << "  cap: " <<
arr.capacity() << endl;

arr.push_back(6); // добавляем другой элемент
cout << "size: " << arr.size() << "  cap: " <<
arr.capacity() << endl;
```

Для добавления элементов в `std::vector` можно использовать ряд функций:

`push_back(val)` - добавляет значение `val` в конец вектора с помощью копирования объекта (или обеспечивает перемещение, если возможно).
`emplace_back(val)` - добавляет значение `val` в конец вектора без копирования или перемещения.
`emplace(pos, val)` - вставляет элемент `val` на позицию, на которую указывает итератор `pos`. Возвращает итератор на добавленный элемент.

`emplace_back` и `emplace` стал возможен благодаря появлению `variadic templates` (вариативные шаблоны) и `forwarding`. По сути это просто отложенный вызов конструктора, который произойдёт внутри контейнера на заранее выделенной памяти, а не вне его, как это происходит с `push_back`.

`insert(pos, val)` - вставляет элемент `val` на позицию, на которую указывает итератор pos, аналогично функции emplace. Возвращает итератор на добавленный элемент.
`insert(pos, n, val)` - вставляет `n` элементов `val` начиная с позиции, на которую указывает итератор `pos`. Возвращает итератор на первый добавленный элемент. Если `n=0`, то возвращается итератор `pos`.
`insert(pos, begin, end)` - вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end. Возвращает итератор на первый добавленный элемент. Если между итераторами begin и end нет элементов, то возвращается итератор pos.
`insert(pos, values)` - вставляет список значений values начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если values не содержит элементов, то возвращается итератор pos.

`std::vector` можно использовать в качестве стека, для этого реализованы три основные функции:

функция `push_back()` — добавляет элемент в стек.
функция `back()` — возвращает значение верхнего элемента стека без удаления.
функция `pop_back()` — вытягивает элемент из стека.

```
vector<int> stack;
stack.push_back(7); // 7
stack.push_back(4); // 7, 4
stack.push_back(1); // 7, 4, 1

stack.pop_back(); // 7, 4
stack.pop_back(); // 7
stack.pop_back();
```

В некоторых случаях, если произошло удаление большого количества элементов из вектора (`size()` уменьшился, а `capacity()` осталась большой), если дальнейшее добавление элементов в вектор не планируется, то логично освободить задействованную память. Для этого есть специальный метод `shrink_to_fit()`.

```
vector <int> vec {1,2,3,5,8,9}; // cap: 6 size: 6
vec.clear(); // cap: 6 size: 0
vec.push_back(8); // cap: 6 size: 1
vec.shrink_to_fit(); // cap: 1 size: 1
```

std::vector<bool>

Подход к реализации `vector<bool>` позволяет сэкономить память. Дело в том, что тип `bool` не может занимать меньше единицы адресуемой памяти, то есть один байт. Однако в реализации вектора хранить получение значения можно в битах, при этом получается восьмикратная экономия памяти. Однако из за такой реализации это контейнер, к объектам которого нет прямого доступа. Работать с ними нужно через специальные прокси-объекты, что обычно приводит к снижению производительности. Если критично время выполнения и не очень критичны затраты памяти, можно использовать `std::vector<std::uint8_t>`.

std::deque

Контейнер `deque` упорядочивает элементы заданного типа в линейном порядке и, подобно векторам, обеспечивает быстрый произвольный доступ к любому элементу и эффективную вставку и удаление в конце контейнера. Для использования данного контейнера нужно подключить заголовочный файл `deque`.

```
std::deque<int> deque1; // пустая очередь
std::deque<int> deque2(5); // deque2 состоит из 5 чисел, каждый элемент имеет значение по умолчанию
std::deque<int> deque3(5, 2); // deque3 состоит из 5 чисел, каждое число равно 2
std::deque<int> deque4{ 1, 2, 4, 5 }; // deque4 состоит из чисел 1, 2, 4, 5
std::deque<int> deque5 = { 1, 2, 3, 5 }; // deque5 состоит из чисел 1, 2, 3, 5
std::deque<int> deque6({ 1, 2, 3, 4, 5 }); // deque6 состоит из чисел 1, 2, 3, 4, 5
std::deque<int> deque7(deque4); // deque7 - копия очереди deque4
std::deque<int> deque8 = deque7; // deque8 - копия очереди deque7
```

Для получения элементов очереди можно использовать операцию `[ ]` и ряд функций:

`[index]`: получение элемента по индексу
`at(index)`: возвращает элемент по индексу
`front()`: возвращает первый элемент
`back()`: возвращает последний элемент
`pop_front()`: удаляет первый элемент из дека, не возвращает значение
`pop_back()`: удаляет последний элемент из дека, не возвращает значение
`push_front(elem)`: добавляет новый элемент `elem` в начало дека
`push_back(elem)`: добавляет новый элемент `elem` в конец дека
`empty()`: возвращает true, если дек пуст, или `false`, если не пуст

Если происходит обращение с помощью операции индексирования по некорректному индексу, который выходит за границы контейнера, то результат будет неопредленным. В этом случае использование функции `at()` является более предпочтительным, так как при обращении по некорректному индексу она генерирует исключение `out_of_range`.

Чтобы узнать размер очереди, можно использовать функцию `size()`. А функция `empty()` позволяет узнать, содержит ли очередь элементы. Она возвращает значение true, если в очереди есть элементы. Функция `resize()` позволяет изменить размер очереди. Эта функция имеет две формы: `resize(n)`: оставляет в очереди n первых элементов. Если `deque` содержит больше элементов, то размер контейнера усекается до первых n элементов. Если размер очереди меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию.

`resize(n, value)`: также оставляет в очереди n первых элементов. Если размер очереди меньше `n`, то добавляются недостающие элементы со значением `value`.

Функция `assign()` позволяет заменить все элементы очереди определенным набором. Она имеет следующие формы:

`assign(il)`: заменяет содержимое контейнера элементами из списка инициализации `il`

`assign(n, value)`: заменяет содержимое контейнера `n` элементами, которые имеют значение `value`

`assign(begin, end)`: заменяет содержимое контейнера элементами из диапазона, на начало и конец которого указывают итераторы `begin` и `end`

```
std::deque<int> numbers = { 1, 2, 3, 4, 5 };
numbers.assign({ 21, 22, 23, 24, 25 }); // numbers = { 21, 22, 23, 24, 25 }
numbers.assign(4, 3); // numbers = {3, 3, 3, 3}
std::deque<int> values = { 6, 7, 8, 9, 10, 11 };
auto start = values.begin() + 2; // итератор указывает на третий элемент
auto end = values.end(); // итератор указывает на последний элемент
numbers.assign(start, end); // numbers = { 8, 9, 10, 11 }
```

Для добавления элементов в очередь `std::deque` можно использовать ряд функций:

Вставка:
`push_back(val), push_front(val), emplace_back(val), emplace_front(val), emplace(pos, val), insert(pos, val), insert(pos, n, val), insert(pos, begin, end), insert(pos, values)`

Удаление:
`clear(p), pop_back(), pop_front(), erase(p), erase(begin, end)`

Контейнеры `vector` и `deque` очень схожи по описанию. Тем не менее, разница между ними есть и весьма существенная. Кроме того, что к `deque` можно добавлять элементы в начало, он отличается от `vector` размещением в памяти. `vector` всегда будет размещаться в памяти последовательно. Из-за последовательного размещения в памяти произвольный доступ к элементам очень быстрый.
Контейнер `deque` может быть сегментирован в памяти. Упрощённо `deque` можно представить как связанный список векторов. Поэтому доступ к элементам будет медленнее, чем у вектора. Однако вектор при расширении может получить новую область памяти в другом месте, при этом произойдёт копирование всех элементов, `deque` гарантированно выделит память под новые элементы без копирования.

std::list

`std::list` — это двусвязный список, контейнер, который поддерживает постоянное время вставки и удаления элементов из любой точки контейнера. Быстрый произвольный доступ не поддерживается. Для `std::list` характерны похожие для всех контейнеров методы вставки и удаления:

Вставка:
`push_back(val), push_front(val), emplace_back(val), emplace_front(val), emplace(pos, val), insert(pos, val), insert(pos, n, val), insert(pos, begin, end), insert(pos, values)`

Удаление:
`clear(p), pop_back(), pop_front(), erase(p), erase(begin, end)`

std::forward_list

`std::forward_list` — это контейнер, который поддерживает быструю вставку и удаление элементов из любой точки контейнера. Быстрый произвольный доступ не поддерживается. Он реализован в виде односвязного списка и, по существу, не имеет никаких накладных расходов по сравнению с его реализацией в C. По сравнению с `std::list` этот контейнер обеспечивает более эффективное хранение, когда двунаправленная итерация не требуется:

Вставка:
`push_front(val), emplace_front(val), emplace_after(p, val), insert_after(p, val), insert_after(p, n, val), insert_after(p, begin, end), insert(pos, values)`

Удаление:
`clear(), pop_front(), erase_after(p), erase_after(begin, end)`

std::stack

Для использования `std::stack` нужно подключить заголовочный файл `stack`. Стек это структура данных, реализующее поведение Last In First Out (FILO). По умолчанию структура данных стек реализована как адаптер над контейнером `std::deque`. Это значит, что методы `std::deque` ограничены таким образом, чтобы соответствовать концепции стека. Не только лишь `std::deque` может быть контейнером, адаптированным под стек — шаблон `std::stack` принимает первым значением тип хранимых данных, а вторым тип контейнера:

```
std::stack<int> sck1; // этот стек адаптирует контейнер std::deque под принцип работы стека

std::stack<int, std::vector<int>> sck2; // а этот стек адаптирует контейнер std::vector под принцип работы стека
```

Методы доступа к стеку:

`push (val)` - добавление элемента в вершину стека
`pop ()` - удаление верхнего элемента
`top ()` - возвращает верхний элемент без удаления
`empty()` - проверка пуст ли стек
`emplace()` - сконструировать элемент внутри стека
`swap(stack)` - позволяет обменять содержимое одного стека на другой
`size` - узнать размер стека

std::queue

Для использования `std::queue` нужно подключить заголовочный файл `queue`. Очередь это структура данных, реализующее поведение First In First Out (FIFO). По умолчанию структура данных очередь реализована как адаптер над контейнером `std::deque`. Это значит, что методы `std::deque` ограничены таким образом, чтобы соответствовать концепции очереди.

```
std::queue<int> queue1; // эта очередь адаптирует контейнер std::deque под принцип работы очереди

std::queue<int, std::vector<int>> queue2; // эта очередь адаптирует контейнер std::vector под принцип работы очереди
```

Методы доступа к очереди:

`push (val)` - добавление элемента в очередь
`emplace(…)` - добавление элемента в очередь без копирования или перемещения
`pop ()` - удаление первого элемента в очереди
`front ()` - возвращает первый элемент
`back ()` - возвращает последний элемент в очереди
`empty()` - проверка пуста ли очередь
`swap(stack)` - позволяет обменять содержимое одного вектора на другой

std::priority_queue

Очередь с приоритетом (`priority_queue`) — это очередь, в которая возвращает элементы отсортированном порядке по возрастанию или убыванию. Функция сравнения задаётся по умолчанию функцией `std::less`, переопределись её можно третим параметром шаблона.

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename
Container::value_type>
> class priority_queue;
```

Стандартные алгоритмы

Поскольку поиск, подсчет и сортировка являются очень распространенными операциями в программировании, то в состав Стандартной библиотеки C++ изначально уже включен большой набор функций, которые выполняют данные задачи всего в несколько строчек кода. В дополнение к этому, эти функции уже предварительно протестированные, эффективные и имеют поддержку множества различных типов контейнеров. Некоторые из этих функций поддерживают и распараллеливание — возможность выделять несколько потоков ЦП для одной и той же задачи, чтобы выполнить её быстрее.

std::find

Функция `std::find()` выполняет поиск первого вхождения заданного значения в контейнере. В качестве аргументов `std::find()` принимает итератор для начального элемента в последовательности, итератор для конечного элемента в последовательности, значение для поиска. В результате будет возвращен итератор, указывающий на элемент с искомым значением, если такой элемент будет найден. Например:

```
std::array<int, 6> arr{ 13, 90, 99, 5, 40, 8};
int search = 90;
std::array<int, 6>::const_iterator result =
std::find(arr.begin(), arr.end(), search);
```

std::find_if

Алгоритм `std::find_if()` и поиск элемента с условием. `std::find_if()` работает аналогично функции `std::find()`, но вместо того, чтобы передавать значение для поиска, передаётся вызываемый объект, например, указатель на функцию, который проверяет, найдено ли совпадение. Функция `std::find_if()` будет вызывать этот объект для каждого элемента, пока не найдет искомый элемент или в контейнере больше не останется элементов для проверки.

```
bool containsNut(const std::string& str){
  return str.find("nut") != std::string::npos;
}

array<std::string, 4> arr{ "apple", "banana",
"walnut", "lemon" };
auto found = std::find_if(arr.begin(), arr.end(),
containsNut);
cout << *found << endl;
```

std::count()/std::count_if()

Функции `std::count()` и `std::count_if()` ищут все вхождения элемента или элемент, соответствующий заданным критериям и возвращает количество. Пример, в котором подсчитывается сколько элементов содержит подстроку `"nut"`:

```
bool containsNut(std::string str){
  return str.find("nut") != std::string::npos;
}

array<string, 5> arr{"apple", "banana",
"walnut", "lemon", "peanut"};
auto found = std::count_if(arr.begin(),
arr.end(), containsNut);
cout << found << endl;
```

std::sort()

`std::sort()` по умолчанию сортирует массив в порядке возрастания, но возможности `std::sort()` этим не ограничиваются. Есть версия `std::sort()`, которая принимает вспомогательную функцию в качестве третьего параметра, что позволяет кастомизировать сортировку. Данная вспомогательная функция принимает два параметра для сравнения и возвращает `true`, если первый аргумент должен быть упорядочен перед вторым. По умолчанию, `std::sort()` сортирует элементы в порядке возрастания.

```
bool greater(int a, int b){ return (a > b); }
…
std::array<int, 6> arr{ 13, 90, 99, 5, 40, 80 };
std::sort(arr.begin(), arr.end(), greater);
for (int i : arr){ std::cout << i << std::endl;}
// >> 99, 90, 80, 40, 13, 5
```

std::for_each()

Функция `std::for_each()` принимает список в качестве входных данных и применяет пользовательскую функцию к каждому элементу этого списка. Это полезно, когда нужно выполнить одну и ту же операцию со всеми элементами списка. Например:

```
void doubleNums(int &i){ i *= 2; }
void printNums(int &i){ cout << i << endl; }
```

```
…

std::array<int, 4> arr{ 1, 2, 3, 4 };
std::for_each(arr.begin(), arr.end(),
doubleNums);
std::for_each(arr.begin(), arr.end(), printNums);
```

std::unique()

Функция std::unique() сдвигает дублирующиеся элементы в конец контейнера. Для корректной работы std::unique() требуется, чтобы контейнер был отсортирован. Например:

```
void printNums(int &i){ std::cout << i << " ";}
```

…

```
vector<int> arr = {10 ,20, 30, 10 ,49, 67, 10};

for_each(arr.begin(), arr.end(), printNums);
cout << endl;

sort(arr.begin(), arr.end());
auto u = unique(arr.begin(), arr.end());
arr.erase( u , arr.end() );

for_each(arr.begin(), arr.end(), printNums);
cout << endl;
```