Использование основных контейнеров STL

Кориненко Матвей

Октябрь 2021

Оглавление

STL — стандартная библиотека шаблонов C++. В ней содержатся готовые контейнеры, алгоритмы и итераторы, которые Вы можете свободно использовать в своих программах.

Мы познакомимся с контейнерами

- stack
- queue
- deque
- set
- map

и некоторыми их модификациями.

Все контейнеры в данном материале будут рассматриваться на примере языка программирования C++.

Глава 1

Классификация контейнеров

Контейнеры в C++ делятся на три класса:

- **Последовательные контейнеры** (к ним из предложенных к рассмотрению относится только deque) — контейнеры, элементы которых находятся в последовательности. Добавлять элемент можно в любое место этого контейнера.
- **Ассоциативные контейнеры** (к ним относятся map и set) — контейнеры, автоматически сортирующие свои элементы по определенному принципу.
- **Адаптеры** (к ним относятся stack и queue) — контейнеры, созданные для определенных задач.

При использовании контейнеров и алгоритмов, работающих с ними, используются **итераторы** — интерфейсы для взаимодействия с элементами контейнера. Каждому элементу контейнера ставится в соответствие свой итератор.

Глава 2

Основные методы работы с контейнерами в C++

2.1 Общие методы для всех контейнеров

Для подключения любого контейнера к Вашему файлу, необходимо использовать следующий код (далее на месте слова «container» нужно использовать наименование того контейнера, который Вы хотите использовать, например, stack).

```
#include<container>
```

Создание одного экземпляра контейнера происходит следующим образом.

```
container<type> c;
// создан контейнер, который может содержать
// элементы типа type
```

Чтобы узнать размер контейнера (количество элементов в нем), нужно использовать метод size, возвращающий целое число — размер контейнера.

```
int sz = c.size();
// в переменной sz теперь хранится размер контейнера c
```

Если имеется два одинаковых контейнера, содержащих элементы одного типа, можно обменять содержимое этих контейнеров местами, используя метод swap.

```
c1.swap(c2);
// теперь в контейнере c1 хранятся элементы из c2,
// а в контейнере c2 - из c1
```

Если нам требуется узнать, пустой контейнер или нет, достаточно просто использовать метод empty, который возвращает булево значение true, если контейнер пустой, иначе — false.

```
bool is_empty = c.empty();
```

Эти методы можно использовать при работе с любыми контейнерами (даже адаптерами) в C++.

2.2 Общие методы для последовательных контейнеров

Если используемый контейнер относится к классу последовательных, его функционал можно расширить с помощью встроенных в STL алгоритмов.

При создании контейнера, можно сразу задать его размер и, если требуется, элемент по умолчанию.

```
container<type> c(n, x);
// будет создан контейнер с размером n,
// каждый элемент которого равен x
```

Например, можно отсортировать элементы контейнера по возрастанию с помощью функции sort за временную сложность $O(n \log n)$, где n — размер контейнера.

```
sort(c.begin(), c.end());
```

Здесь c.begin() и c.end() — итераторы, задающие полуинтервал, на котором будет происходить сортировка.

Другими словами, c.begin() — итератор, указывающий на первый элемент контейнера, c.end() — итератор, указывающий на элемент, следующий за последним элементом контейнера. Поэтому если нам захочется отсортировать, например, все элементы кроме первых k, мы можем воспользоваться следующим кодом.

```
sort(c.begin() + k, c.end());
```

Используя итераторы, можно «пробегаться» по всем элементам контейнера (как последовательного, так и ассоциативного), используя цикл. Чтобы получить элемент по итератору, используется оператор *.

```
for (auto it = c.begin(); it < c.end(); it++) {
    type x = *it;
}
```

«Перевернуть» последовательную часть контейнера можно, используя функцию reverse.

```
reverse(c.begin(), c.end() - 3);
```

Теперь элементы контейнера c, кроме последних трех, идут в обратном порядке.

Если наш контейнер уже имеет какой-то размер, можно заполнить его элементами одного и того же значения.

```
fill(c.begin(), c.end(), x);
// значение x будет присвоено всем элементам
```

Для поиска минимального и максимального элемента последовательного контейнера, можно воспользоваться функциями min_element и max_element.

```
auto it_max = max_element(c.begin(), c.end());
// в it_max хранится итератор максимального элемента
auto it_min = min_element(c.begin(), c.end());
// в it_min теперь находится итератор минимального элемента
```

А если нам потребуется очистить контейнер, можно использовать метод clear.

```
c.clear();
```

2.3 Общие методы для ассоциативных контейнеров

Поскольку ассоциативные контейнеры умеют автоматически сортировать свои элементы при удалении (добавлении) новых, операции с ними работают достаточно быстро.

Функции добавления, удаления, поиска элемента в ассоциативном контейнере работают с временной сложностью $O(\log n)$, где n — размер контейнера.

Если нам потребуется проверить наличие элемента x в контейнере, можно использовать метод find, который вернет итератор элемента x. А если его нет в контейнере, вернет итератор конца контейнера.

```
auto it = c.find(x);
// если в контейнере нет элемента x, то
// it = c.end()
```

Если нам понадобится удалить из контейнера элемент, мы можем это сделать двумя способами: удалить по итератору или удалить по значению. При удалении по итератору всегда удаляется только один элемент, потому что одному итератору всегда соответствует один элемент. Но если в контейнере содержатся равные элементы (например, в multiset), то при удалении по значению x, удалятся все элементы с этим значением.

```
c.erase(x);
// если x - значение элемента, удалятся
// все элементы с этим значением

c.erase(it);
// если it - итератор, удалится только один элемент,
// которому соответствует этот итератор
```

Для поиска минимального или максимального элемента в ассоциативном контейнере также существуют функции min_element и max_element, возвращающие итератор на соответствующий элемент.

```
auto it_max = max_element(c.begin(), c.end());
// в it_max хранится итератор максимального элемента
auto it_min = min_element(c.begin(), c.end());
// в it_min теперь находится итератор минимального элемента
```

Для получения количества элементов контейнера, равных x, можно воспользоваться функцией count, возвращающей целое число.

```
int count_x = c.count(x);
// в count_x будет храниться количество
// элементов контейнера, равных x
```

Ассоциативные контейнеры, как и последовательные, можно очистить с помощью метода clear.

```
c.clear();
// теперь контейнер пустой
```

Глава 3

Контейнер stack

3.1 Описание контейнера

Стек — структура данных, работающая по принципу FILO (first-in-last-out, первым-зашел-последним-вышел). Абстрактно представить этот контейнер можно в виде груды посуды — следующую тарелку Вы кладете наверх, но и доставать потом тоже нужно сверху.

При использовании этой структуры, Вы имеете доступ только к последнему добавленному элементу. Вы можете посмотреть на него, извлечь, или поверх него положить новый.

Операции добавления и удаления последнего элемента из стека, просмотра последнего элемента работают с временной сложностью $O(1)$, то есть никак не зависят от количества элементов внутри него.

Стек является контейнером-адаптером, поэтому не имеет каких-либо дополнительных функций кроме тех, для которых он предназначен.

3.2 Использование контейнера

Создадим стек, содержащий элементы типа int.

```
stack<int> st;
```

Чтобы добавить первый элемент в стек, необходимо использовать метод push.

```
st.push(5);
```

Теперь в нашем стеке хранится число 5. Давайте добавим еще несколько чисел в него.

```
st.push(6);
st.push(1);
```

Если нам понадобится узнать верхний элемент стека, используем метод top, который вернет значение того же типа данных, что и содержимое в стеке.

```
int stack_top = st.top();
// stack_top = 1
```

Для удаления верхнего элемента стека используем метод pop. Если его использовать в случае пустого стека, ничего не произойдет.

```
st.pop();
// теперь на вершине стека находится число 6
```

Глава 4

Контейнер queue

4.1 Описание контейнера

Очередь — структура данных, работающая по принципу FIFO (first-in-first-out, первым-зашел-первым-вышел).

При работе с этим контейнером Вы имеете доступ к первому и последнему элементу очереди. Первый элемент можно осмотреть и извлечь, последний — осмотреть и добавить новый. Доступ к последнему элементу очереди будет возможен, когда вся очередь перед ним очистится.

Операции с очередью, аналогично стеку, работают с временной сложностью $O(1)$, то есть никак не зависят от количества элементов в контейнере.

Очередь тоже является контейнером-адаптером, поэтому кроме своих собственных методов, она не имеет ничего.

4.2 Использование контейнера

Очередь может также свободно содержать элементы любого фиксированного типа данных, пусть в этот раз это будут символы.

```
queue<char> q;
```

Добавление элемента в очередь происходит с использованием метода push, аналогично стеку.

```
q.push('a');
```

Добавим еще несколько элементов.

```
q.push('b');
q.push('c');
```

Первый и последний элементы в очереди просматриваются с помощью соответственно методов front и back, которые возвращают элементы того же типа, что и содержимое очереди.

```
char queue_front = q.front();
// queue_front = 'a'
char queue_back = q.back();
// queue_back = 'c'
```

С помощью метода pop можно извлечь первый элемент в очереди.

```
q.pop();
```

Теперь первым элементом очереди является символ 'b'.

Глава 5

Контейнер deque

5.1 Описание контейнера

Deque (double-ended-queue, очередь-с-двумя-концами) — структура данных, позволяющая добавлять элементы как в начало, так и в конец контейнера, удалять их, обращаться по индексу (итератору), изменять.

Операции добавления элемента в начало и конец контейнера, просмотра и изменения элемента работают с временной сложностью $O(1)$.

Обощая, контейнер deque можно использовать как более функциональную замену queue и stack, так как он еще и относится к последовательным контейнерам.

5.2 Использование контейнера

Для начала создадим экземпляр deque, который может содержать целые числа.

```
deque<int> d;
```

Добавление элемента в конец deque осуществляется с помощью метода push_back, в начало — с помощью push_front.

```
d.push_front(4);
d.push_back(3);
d.push_back(1);
d.push_front(2);
```

Сейчас элементы deque в порядке от начала до конца — {2, 4, 3, 1}.

Чтобы обратиться к первому элементу deque, можно использовать индекс 0 (как в массивах) или метод front.

```
int front = d.front();
// front = 2
int front_index = d[0];
// front_index = 2
```

Чтобы обратиться к последнему элементу deque, мы так же можем использовать его индекс (который можно узнать с помощью метода size) или метод back.

```
int back = d.back();
// back = 1
int back_index = d[d.size() - 1];
// back_index = 1
```

Для удаления элемента из начала deque используется метод pop_front, из конца — pop_back.

```
d.pop_back();
d.pop_front();
```

Теперь в deque содержатся только два элемента — {4, 3}.

В deque можно изменить элемент по его индексу, как в обычном массиве.

```
d[0] = 2;
// теперь deque состоит из элементов {2, 3}
```

Глава 6

Контейнер set

6.1 Описание контейнера

Set — структура данных, хранящая свои элементы в отсортированном порядке в единственном экземпляре. Эту структуру называют множеством.

Множество поддерживает операции добавления, поиска, удаления элементов. Они работают с временной сложностью $O(\log n)$, то есть зависят от размера контейнера. Но для регулярно используемых значений n в задачах, это все еще очень быстро.

6.2 Использование контейнера

Создадим экземпляр множества, содержащий целые числа. Для создания множества из нестандартных структур, необходимо в этих структурах определить оператор «<», либо использовать компаратор, так как множество должно быть упорядочено.

```
set<int> st;
```

Для добавления элемента в множество используется метод insert.

```
st.insert(1);
st.insert(2);
st.insert(3);
st.insert(3);
```

Сейчас мы добавили в множество элементы {1, 2, 3, 3}; но поскольку множество содержит только уникальные элементы, его размер будет 3 и состоять оно будет из элементов {1, 2, 3}.

Для нахождения элемента множества, **большего либо равного** заданному, существует метод lower_bound, возвращающий итератор на удовлетворяющий элемент.

```
int lb = *st.lower_bound(1);
// lb = 1, т.к. это первый элемент множества,
// больше либо равный 1
```

Для нахождения элемета, **строго большего** заданного, используется метод upper_bound, также возвращающий итератор.

```
int ub = *st.upper_bound(1);
// ub = 2, т.к. это первый элемент множества,
// больший единицы
```

Удаление элемента из множества осуществляется с помощью метода erase.

```
st.erase(3);
// теперь в множестве хранятся только элементы 1 и 2
```

6.3 Модификации контейнера

6.3.1 Контейнер unordered_set

Если нам не важна упорядоченность элементов множества, а необходимо только проверять их наличие в нем, можно использовать unordered_set. Проверка наличия элемента там осуществляется с временной сложностью $O(1)$, то есть не зависит от размера контейнера. Поиск же, например, lower_bound, может выполняться за $O(n)$, поэтому для таких целей unordered_set использовать не рекомендуется.

Для работы этого контейнера необходим hasher, преобразовывающий элемент множества в число (хэш) для быстрой проверки его наличия. Для стандартных типов в C++ hasher писать и добавлять не требуется, но если Вы используете как элементы сложные типы или контейнеры, без хэшера unordered_set работать не будет.

6.3.2 Контейнер multiset

Бывают ситуации, когда нам нужно хранить не только сам элемент, но и знать его количество вхождений в контейнер. Для этого отлично подходит multiset.

Если выполнить следующий код для multiset

```
multiset<int> mst;
mst.insert(1);
mst.insert(2);
mst.insert(3);
mst.insert(3);
```

размер контейнера будет не 3, как в случае с set, а 4.

К тому же, количество вхождений числа 3 в multiset будет 2.

```
int cnt = mst.count(3);
// cnt = 2
```

Чтобы удалить один экземпляр элемента из multiset, можно воспользоваться поиском этого элемента через метод find, который возвращает итератор, и использовать удаление по итератору.

```
mst.erase(mst.find(3));
cnt = mst.count(3);
// cnt = 1
```

Глава 7

Контейнер map

7.1 Описание контейнера

Map — ассоциативный контейнер, работающий по принципу <ключ, значение>. Map также автоматически сортируется по возрастанию ключа. Ключом и значением может быть что угодно.

Функции добавления, изменения, удаления элементов с map работают за временную сложность $O(\log n)$.

7.2 Использование контейнера

Поскольку можно создать map с любыми ключами и любыми значениями, давайте сделаем map с ключом «строка» и значением по нему будет целое число. Строку предварительно тоже нужно будет подключить к файлу, так как она является контейнером. Для строки не нужно писать компаратор, потому что он уже определен в C++, для сложных структур он может потребоваться так же, как и в случае с set.

```
map<string, int> mp;
```

Чтобы добавить элемент с заданным ключом, воспользуемся следующим кодом.

```
mp["Apple"] = 5;
```

Теперь по ключу "Apple"мы получим значение 5. Чтобы его изменить, нужно использовать методы, которые присущи тому типу данных, которой используется у значения по ключу в map.

```
mp["Apple"] += 1;
// mp["Apple"] = 6;
```

Добавим еще один элемент в map.

```
mp["Banana"] = 7;
```

В map тоже можно использовать методы lower_bound и upper_bound, которые будут искать итератор подходящего элемента, опираясь на его ключ. Но в этом случае они будут возвращать итератор на **пару** вида <ключ, значение>. Поэтому сначала необходимо получить элемент map по итератору, а потом из него получить ключ или значение.

```
string key = (*mp.lower_bound("B")).first;
// key = "Banana"
int value = (*mp.lower_bound("B")).second;
// value = 7
```

Удаление в map происходит по ключу. Удаляется весь элемент вида <ключ, значение>.

```
mp.erase("Apple");
```

7.3 Модификации контейнера

7.3.1 Контейнер unordered_map

Зачастую, при использовании map, нам не важен порядок элементов в нем. Нужно лишь уметь сопоставлять ключ со значением по нему. Для этого подходит unordered_map, сложность работы которого для этих задач не зависит от размера контейнера.

Для работы с ним в некоторых случаях может потребоваться так называемый hasher. Но для всех простых ситуаций, когда ключом является, к примеру, число или строка, hasher не нужен.

7.3.2 Другие модификации

Существуют другие вариации этого контейнера, такие как unordered_multimap и multimap, но они используются достаточно редко.

Глава 8

Дополнительные материалы

8.1 Использование своего компаратора

Есть случаи, когда в таких ассоциативных контейнерах как map и set хранятся элементы, которые необходимо сортировать особым образом. Это может быть вынужденная мера (C++ не умеет сортировать элементы такого типа) или необходимость.

Правило сортировки задается с помощью компаратора. Это структура, содержащая булеву функцию, которой на вход подается два элемента одного типа. Функция возвращает true, если по итогам сравнения первый из этих элементов должен стоять в контейнере раньше второго. Иначе она возвращает false.

Вот пример создания и подключения компаратора (для сортировки пар по убыванию первого элемента и возрастанию второго).

```
struct cmp {
    bool operator() (pair<int, int> a, pair<int, int> b) {
        if (a.first > b.first)
            return true;
        else if (a.first == b.first)
            return a.second < b.second;
        else
            return false;
    }
};
set<pair<int, int>, cmp> st;
```

8.2 Использование своего хэшера

При использовании unordered_set и unordered_map для быстрого доступа к элементу используется хэш-функция. Преобразование элемента в число позволяет в некоторых моментах существенно ускорить поиск в контейнере. Но, как и с компаратором, не все элементы C++ умеет преобразовывать в число. Для этого нужно написать свой хэшер — класс, содержащий функцию, которая при получении элемента заданного типа вернет уникальное число, благодаря которому будет осуществляться быстрый поиск в контейнере. Число должно быть **уникальным**, потому что если это будет не так, то разным элементам может соответствовать один хэш, из-за чего работа с контейнером будет некорректной (например, подсчет количества одинаковых элементов в unordered_multiset).

Вот пример хэшера для пары чисел, находящихся в пределах от 0 до 99.

```
class pairHash{
    public:
        size_t operator() (const pair<int, int> &k) const {
            return k.first * 100 + k.second;
        }
};
unordered_map<pair<int, int>, int, pairHash> mp;
```