

# 容器与迭代器

IAD

CS@NJU

2025 年 3 月 21 日

## 1 基本容器

# 什么是容器

能「以一定规则」去「装一些东西」的东西

- 1 以什么规则：数据结构（数组、栈、队列、双端队列、链表.....）
- 2 装什么东西：什么都能装（int, float, struct, Container.....）
- 3 有什么容器：
  - 1 顺序型：vector（可变长度数组）、deque（双端队列）、list（链表）、array（数组）
  - 2 关系型：map/unordered\_map/multimap（映射）、set/unordered\_set/multiset（集合）
  - 3 适配器：queue（队列）、stack（栈）、priority\_queue（堆）
- 4 容器的基本属性
  - 1 大小：`.size()` 返回容器里的元素个数
  - 2 判空：`.empty()` 返回 true/false 表示容器是否是空的
  - 3 迭代器：`.begin()` `.end()`

对容器有任何疑问，去翻：<https://zh.cppreference.com/>

# vector: 可变长度数组

初始长度为 0, 使用 `push_back` 向末尾添加元素。

- 1 定义: `vector<int> a;`
- 2 `a.push_back(45); // 45`
- 3 `a.push_back(67); // 45 67`
- 4 `a.push_back(89); // 45 67 89`
- 5 `cout<<a[1]; // Output: 67`
- 6 `a.push_back("This is a string"); // Error`

# vector: 到底能装什么?

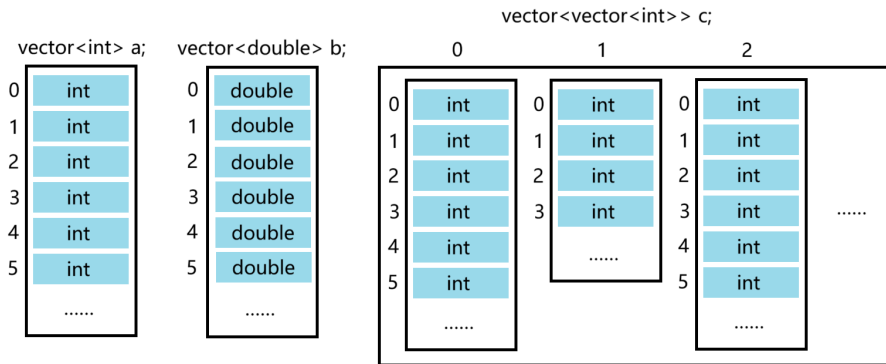


图: vector

# deque: 双端队列

一根长管子，可以从两端放入/拿取。

- 1 获取最前面的元素: `dq.front()`
- 2 获取最后面的元素: `dq.back()`
- 3 向最前面插入: `dq.push_front(u)`
- 4 向最后面插入: `dq.push_back(u)`
- 5 从最前面弹出: `dq.pop_front()`
- 6 从最后面弹出: `dq.pop_back()`
- 7 找到第  $k$  个元素: `dq[k]`

练习: `deque_vector.cpp`

指向容器中元素的指针。

默认提供的迭代器：

- 1 `.begin()` 指向开头第一个元素
- 2 `.end()` 指向最后一个元素的后一位（此处不存在元素）

移动迭代器：

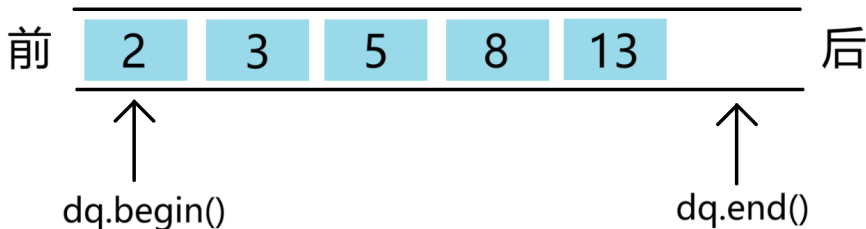
不同的容器支持的操作不同，但基本都支持以下两种：

- 1 `++` 迭代器向后移动一格
- 2 `--` 迭代器向前移动一格

# 迭代器：以 deque 为例

初始 deque

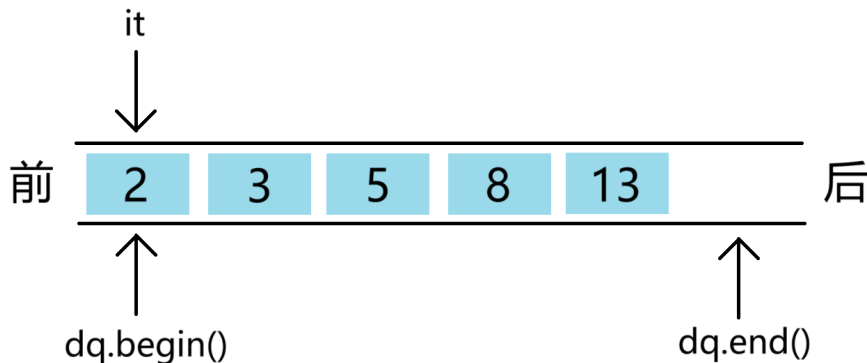
```
deque<int> dq;
```



# 迭代器：以 deque 为例

```
deque<int>::iterator it = dq.begin();
```

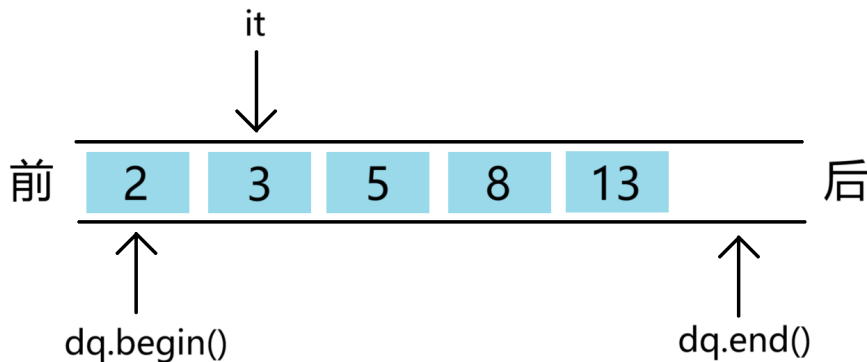
```
deque<int> dq;
```



# 迭代器：以 deque 为例

```
it++;
```

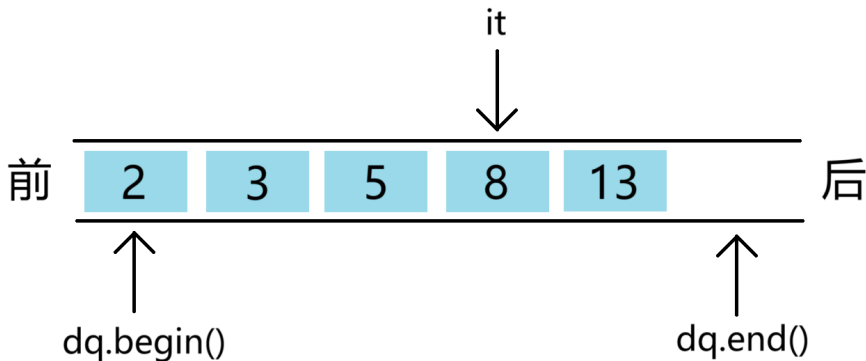
```
deque<int> dq;
```



# 迭代器：以 deque 为例

```
it += 2;
```

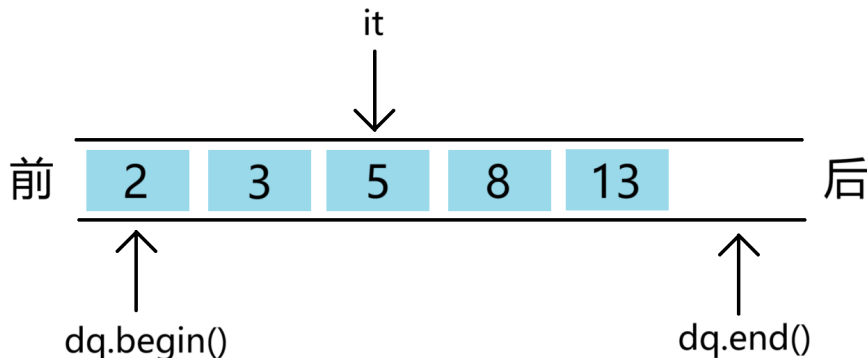
```
deque<int> dq;
```



# 迭代器：以 deque 为例

```
it -= 1;
```

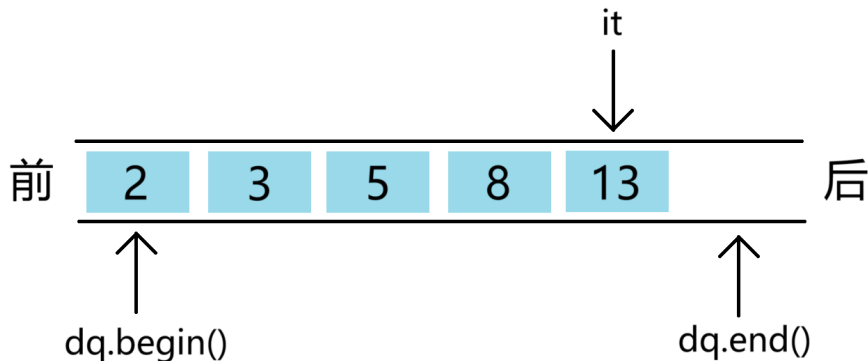
```
deque<int> dq;
```



# 迭代器：以 deque 为例

```
it = dq.end() - 1;
```

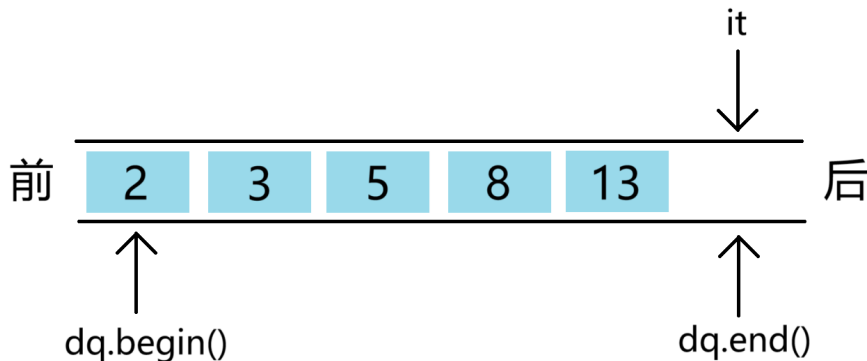
```
deque<int> dq;
```



# 迭代器：以 deque 为例

```
it += 1; // 此时有 it == dq.end()
```

`deque<int> dq;`



# 迭代器的类型

在被指向容器的类型后面，加上 `::iterator` 即可。

- 1 `vector<int>::iterator vec_it;`
- 2 `deque<double>::iterator dq_it;`
- 3 `deque<vector<int> >::iterator dq_it;`

当然，也可以用 `auto`（为完成类型推导，此时必须指定初始值）：

- 1 `auto vec_it = vec.begin();`
- 2 `auto dq_it = dq_vec.end();`

# 迭代器的取值

回顾：迭代器是指向容器中元素的**指针**。

因此与指针类似，当拥有迭代器 `it` 时，取值需要使用 `*it`。

举例：将一个 `deque<double> dq` 的正数第二项改为 3.15

```
1 deque<double>::iterator it = dq.begin();  
2 it ++;  
3 *it = 3.15;
```

举例：输出一个 `vector<int> vec` 的所有元素

```
1 for(auto it = vec.begin(); it != vec.end(); it++)  
2     cout << *it << " ";
```

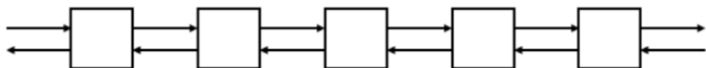
练习：A 题

一个链表, 和 deque 基本一样, 除了: **不支持直接查找第 k 个元素**

仍然支持:

- 1 `.front()`, `.back()`
- 2 `.push_back()`, `.push_front()`
- 3 `.pop_back()`, `.pop_front()`

list:



forward\_list:



# 使用迭代器删除元素

大部分容器支持：删除迭代器所指向的元素 `.erase(it)`

例子：删除一个 `deque<int> dq;` 的最后一个数：

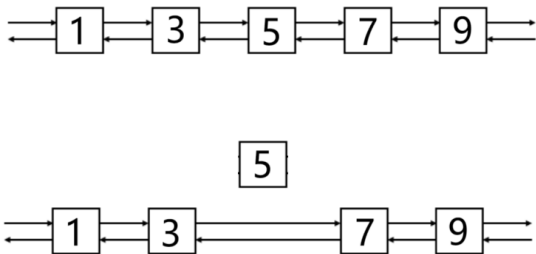
- 1 `deque<int>::iterator it = dq.end();`
- 2 `it -= 1;`
- 3 `dq.erase(it);`
- 4 或者直接：`dq.erase(dq.end() - 1);`

思考：删除一个 `list<int> li;` 中的第一个 5

错误示范：`wrongdel.cpp`

# 为什么错误?

原因：执行完 `li.erase(it);` 之后，当前的 `it` 就已经被扔掉了；再执行 `it++` 时显然错误。



正确的做法：`it = li.erase(it);`，此时 `it` 指向被删除元素在删除之前的下一个元素。

练习：B 题

一个**自动保持有序**的集合，保证其中**元素不重复**

```
1 set<int> s;  
2 s.insert(5); // 5  
3 s.insert(2); // 2 5  
4 s.insert(5); // 2 5  
5 s.insert(3); // 2 3 5  
6 cout << *s.begin() << " " << *(s.end() - 1); // Output: 2 5
```

从 set 中查找元素: `set<int>::iterator it = s.find(5);`

- 1 如果找到了，返回指向目标元素的迭代器
- 2 如果没找到，返回 `s.end()`

练习：维护一个姓名名单，查找一个名字是否在名单中

一个**有序的**映射表，用于键值对（key-value）的存储和查询。

例子：按照姓名存储身高：

```
1 map<string, double> height;  
2 height["XiaoMing"] = 171.5  
3 height["XiaoHong"] = 164.1  
4 cout << height["XiaoHong"]; // Output: 164.1  
5 cout << height["IAD"]; // Output: 0
```

与 set 相同，map 也可以使用 `.find()` 查找元素是否存在。

练习：C 题

基于 deque 的包装器，功能是 deque 的子集。

- 1 queue (队列): 前面只能出, 后面只能进
- 2 stack (栈): 只有前面能进出, 后面被封死

<https://blog.csdn.net/kdjjdjjeje128/article/details/128532611>

练习: D 题

# priority\_queue: 优先队列 (堆)

一个自动保持有序的堆，支持：插入、取出顶部元素、弹出顶部元素。

**“自动保持有序的堆”**：顶部元素永远是堆里最大的那个

```
1 priority_queue<int> pq;  
2 pq.push(3); // pq.top() == 3  
3 pq.push(5); // pq.top() == 5  
4 pq.push(1); // pq.top() == 5  
5 pq.pop(); // pq.top() == 3  
6 pq.pop(); // pq.top() == 1  
7 pq.pop(); // pq.empty() == true
```

# 大根堆/小根堆

大根堆：顶部元素永远是最大值；小根堆：顶部元素永远是最小值

priority\_queue 创建出来默认是大根堆

1 创建大根堆：priority\_queue<int> pq;

2 创建小根堆：

```
priority_queue<int, vector<int>, greater<int> > pq;
```

3 这样也是大根堆：

```
priority_queue<int, vector<int>, less<int> > pq;
```

练习：房间里一开始没有人。实现三个功能：

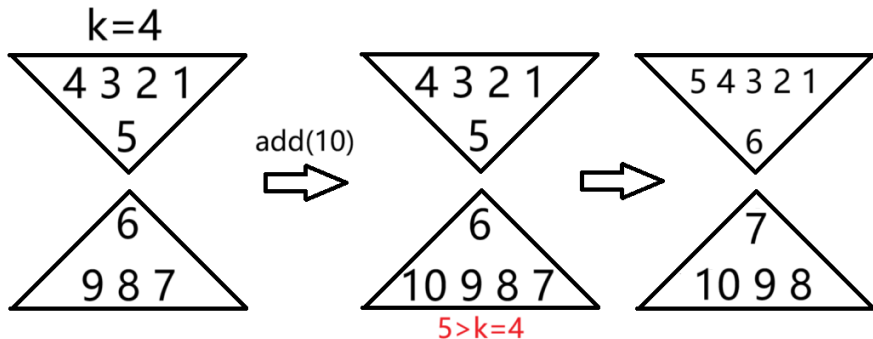
1 void getin(int id); 表示进来了一个学号为 id 的人

2 void getout(); 表示学号最大的人走了

3 int who(); 返回：房间里最大的学号

# 对顶堆

一个小根堆 + 一个大根堆；下面的堆保有不超过  $K$  个元素。



练习：E 题

- 1 Email: 221502001@smail.nju.edu.cn
- 2 WeChat: I\_Am\_Danny\_CN