# Morden cpp notes — container

Monday, December 7, 2020 4:00 PM

容器，就是能够容纳和存放元素的一些数据结构.

- 容器的通用特性：保存元素采用的是"值" value 语义，存储的是元素的拷贝，副本，不是引用. 所以，容器操作元素的很大一块成本是值的拷贝，如元素多，拷贝开销高.

  一个解决办法是用转移赋值函数 std::move().

```
Point p; // 一个拷贝成本很高的对象.
v.push_back(p); // 存储对象，拷贝构造，成本很高.
v.push_back(std::move(p)); // 定义转移构造后就可以转移存储，降低成本.
```

也可使用 C++11 为容器新增的 emplace 操作函数，可以就地构造元素，免去了构造后再拷贝，转移的成本.

```
v.emplace_back(...); // 直接在容器里构造元素，不需要拷贝或者转移.
```

不建议在容器中存放指针，因为不能利用容器自动销毁元素的特性，有内存泄漏隐患. 实在不行，可以用 shared_ptr.

- 容器的特性 {
  - 顺序容器
  - 有序容器
  - 无序容器.

- 顺序容器：线性表：array, vector, deque, list, forward_list.

存储结构 {
- 连续存储的数组：array, vector, deque.
- 指针结构的链表：list, forward_list.

array, vector 对应 C 的内置数组，内存布局与 C 完全兼容，是开销最低，速度最快的容器. 它们的区别在于容量能否动态增长，array 是静态数组，大小在初始化的时候就固定了，而 vector 是动态数组，可以增长.

```
array<int, 2> arr; // 初始化一个array, 长度是 2.
assert(arr.size() == 2); // 静态数组长度总是2.
vector<int> v(2); // 初始一个 vector, 长度为2.
for(int i=0; i<10; i++){
    v.emplace_back(i); // 追加多个元素.
}
assert(v.size() ==12); // 长度动态增长到 12.
```

deque 也是一种可以动态增长的数组，它和 vector 的区别是，可以在两端高效地插入删除元素，所以叫 double-end quene. 而 vector 只能 push_back 在末端追加元素.

```
deque<int> d; // 初始化一个 deque, 长度是 0
d.emplace_back(9); // 末端添加元素.
d.emplace_front(1); // 前端添加一个元素.
assert(d.size() == 2);
```

vector 和 deque 里的元素，因为是连续存储的，所以在中间的插入和删除效率就很低，而 list 和 forward_list 是链表结构，在任意位置插入和删除都很高效.

链表的缺点是查找效率低，只能沿着指针顺序访问，不如 vector 随机访问的效率高. list 是双向链表，可以向前或者向后遍历，而 forward_list 是单向链表，只能向前遍历，查找效率更低.

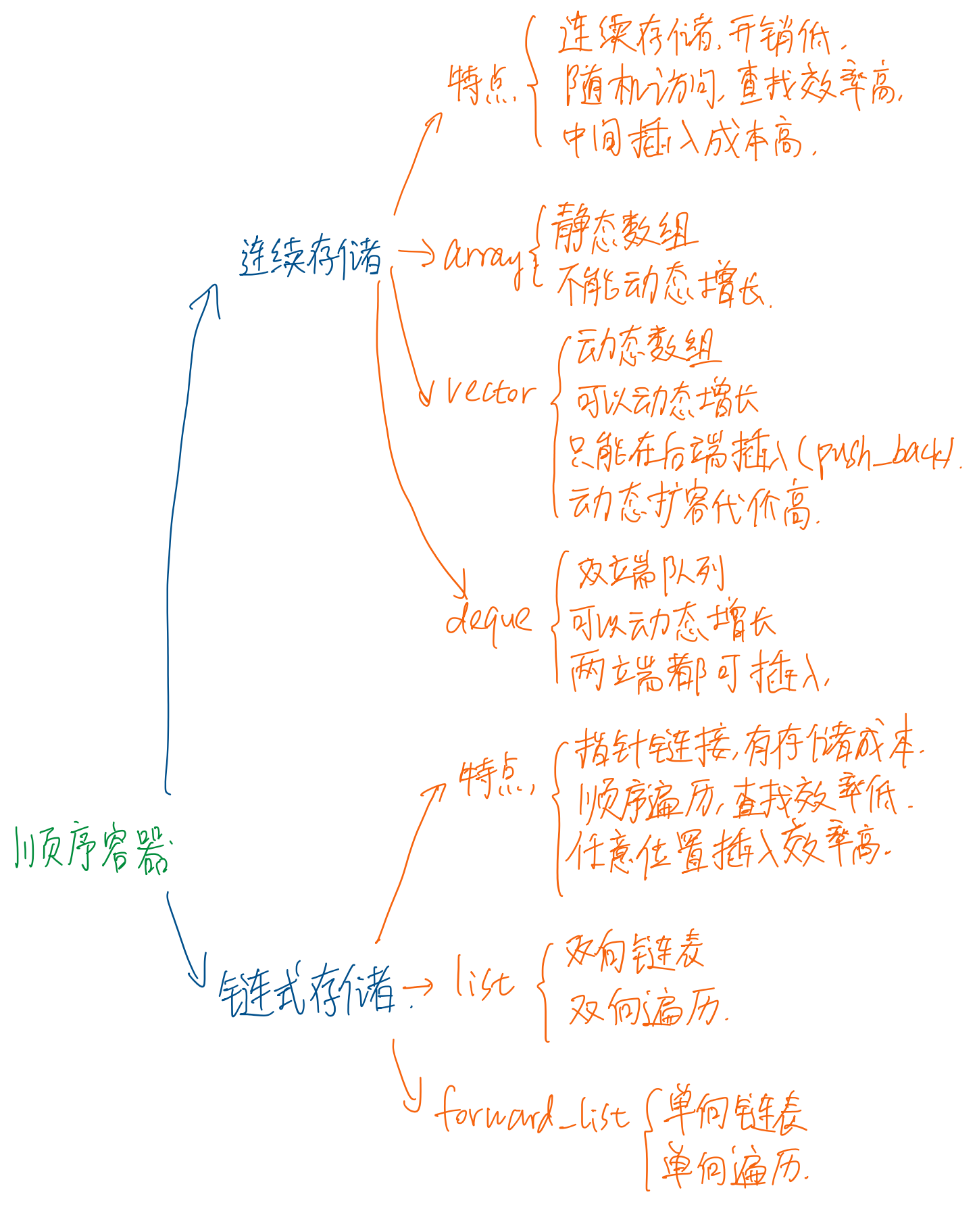
链表结构比起数组还有一个缺点，就是存储成本略高，因为必须为每个元素附加一个或两个指针.

vector/deque 和 list/forward_list 都可以动态增长来容纳更多元素，但内部扩容机制不同.

- 当 vector 容量达到上限，它会再分配一块两倍大小的内存，然后把旧元素拷贝或者移动过去. 这个操作成本很大，在使用 vector 的时候最好预估容量，使用 reserve 提前分配足够空间，减少动态扩容的拷贝代价.
- deque, list 只会按照固定的步长，如 N 个字节，一个节点去扩容，但在短时间插入大量数据的时候就会频繁分配内存，效果反而不如 vector 一次分配.

In general, should use array and vector, because they are fast, easy to use, and easy to sort and find.

deque, list, forward_list can be used when we need special insertion/deletion, if space complexity is of concern, then need to use deque.



- 有序容器：

顺序元素的特点是，元素的次序跟插入次序相同，而在有序容器中，元素插入后就按照，某种规则自动排序，所以是"有序"的.

C++ uses red-black tree, which is the best data structure for find. There are four ordered containers:

- set/multiset，set 是集合
- map/multimap. map 是关联数组(字典).

"multi" means it can hold repeated keys.

有序容器和顺序容器的一个根本区别，是在定义容器的时候必须指定 key 的比较函数. 通常是默认的 less.

```
template<
    class T    // 模板参数只有一个元素类型
> class vector;  ↖ vector

template<
   class key,
   class Compare = std::less<key>
> class set; ← set
```

模板参数是 key 类型，即元素类型. 比较函数是 std::less<key>.

```
template<
    class key,  key类型
    class T,    元素类型
    class Compare = std::less<key> 比较函数.
> map; ← map
```

第一个模板参数是 key 类型，第二个模板参数是元素类型

比较函数是 std::less<key>

C++ 里的 int, string 等基本类型都支持比较排序，但很多自定义类型没有比较函数，解决方法有：

- 重载 <
- 自定义模板参数.

Eg. There's a point class, we need to overload "<" operator, before we can put it in an ordered container:

```
bool operator<(const Point &a, const Point& b)
{
   return a.x<b.x; // Customized comparator
}
set<Point> s;
s.emplace(7);   // now we can put it in container.
```

The Point class is:

```
class Point final
{
Public:
   int x = 0;

public:
   Point(int a) noexcept : x(a)
   {
     cout << "ctor" << x << endl;
   }

   point() noexcept
   {
    cout << "ctor" << x << endl;
   }

   ~Point() noexcept
   {
    cout << "dtor" << x << endl;
   }

   Point(const Point& p) noexcept
   {
     x=p.x;
    cout << "copy ctor" << x << endl;
   }

   Point (Point&& p) noexcept
   {
     x = std::move(p.x);
     cout << "move ctor" << x << endl;
   }
};
```

&与&&区别：

- & 是左值引用，左值指的是在内存中存在的值，不是1这种常量. 左值可以出现在等号左边，也可以出现在等号右边.
- && 是右值引用，rvalue 不能出现在等号左边.

```
Eg. int a = 1;
    int b = 0;
    int c = 3;
    a = b+c;
```

b+c 在执行完后被销毁，不能放在等式左边.

Another way is to use function object or lambda expression, then use that in the container's template parameters, this way is more flexible.

```
set<int> s = {7,3,9}; // define set and initialize with three elements.
for( auto& x: s){
  cout << x << ",";     // output 3,7,9
}
auto comp = [](auto a, auto b)  //define a lambda to compare elements.
{
  return a>b;
};
set<int, decltype(comp)> gs(comp)  // use decltype to get type of lambda.
std::copy(begin(s), end(s),           // copy the elements
          inserter(gs, gs.end()));    // use inserter.
for (auto& x:gs){
  cout << x << ",";        // output 9,7,3.
}
```

std::inserter constructs an insert iterator that inserts new elements into x in successive locations starting at the position pointed by it.

Syntax:

std::inserter( Container & x, typename Container::iterator it);

x: Container in which new elements will be inserted.

it: iterator pointing to the insertion point.

Returns: An insert_iterator that inserts elements into x at the position indicated by it.

```
// C++ Program to demonstrate std::inserter.
#include <iostream>
#include <iterator>
#include <deque>
#include <algorithm>
using namespace std;
int main()
{
   //Declaring first container
   deque<int> v1 = {1,2,3};

   //Declaring second container for copying values
   deque<int> v2 = {4,5,6};

   deque<int>::iterator i1;
   i1 = v2.begin() +1;
   // i1 points to next element of 4 in v2

   // Using std::inserter inside std::copy
   std::copy(v1.begin(), v1.end(), std::inserter(v2, i1));
   // v2 now contains 4,1,2,3,5,6.

   return 0;
}
```

How is inserter helpful?

Inserting value anywhere, if we have to copy value into a container such as a vector, firstly, we had to move elements and then copy, but with the help of std::insert() we can insert at any position with ease.

insert() vs inserter(): when you have to pass an iterator in the algorithm, then you should use inserter(), while for normally inserting the values in the container, insert() can be used.

In place of using std::inserter, we can use insert_iterator, as eventually, std::inserter returns an insert_iterator:

```
// C++ program to demonstrate insert_iterator
#include <iostream>
#include <iterator>
#include <deque>
#include <algorithm>
using namespace std;
int main()
{
  deque<int> v1 = {1, 2, 3};
  deque<int> v2 = {4,5,6};

  deque<int>::iterator ii;
  ii = v2.begin()+1;   // ii points after 4 in v2.

  //Declaring an insert_iterator
  std::insert_iterator<std::deque<int> > i1(v2, ii);

  std::copy(v1.begin(), v1.end(), i1);
  //v2 now contains 4 1 2 3 5 6

  return 0;
}
```

Let's look into an implementation of the STL.

```
_Self& operator=(const typename _Container::value_type& __val){
   _M_iter = container->insert(_M_iter, __val);
   ++_M_iter;
   return *this;
}
```

_M_iter is the position that was passed to std::inserter,

- the position is refreshed by using the one returned by the insert method, insert returns the position of the inserted element, this ensures to always keep _M_iter in the vector's current buffer, even if it was just reallocated.
- the position is incremented, this way each inserted element is positioned right after the previous one. This ensures to have the elements inserted in the right order.

有序容器中，集合关系就用 set，关联数组就用 map. 因为在插入的时候会自动排序，有隐含的插入排序成本，当数据量很大时，内部的位置查找，树旋转成本较高. 如果需要实时插入排序，可以选择 set/map. 如果是非实时，最好还是用 vector，全部数据插入完再一次性排序，效果更好.

- 无序容器：

unordered_set/unordered_multiset

unordered_map/unordered_multimap.

与有序容器区别在于内部数据结构，不是红黑树，是散列表. (哈希表 hash table).

```
using map_type = unordered_map<int, string>;  // 类型别名.
map_type dict;
dict[1] = "one";
dict.emplace(2, "two");
dict[10] = "ten";
for(auto& x: dict){
  cout << x.first << "=>"
       << x.second << ",";
}
```

输出顺序不确定.

对 key 的要求：

```
template<
   class key,       第一个模板参数是key类型
   class T,         第二个模板参数是元素类型
   class Hash = std::hash<key>,   计算散列值的函数对象.
   class keyEqual = std::equal_to<key>  相等比较函数.
> class unordered_map;
```

它要求 key 具备两个条件：

1. 可以计算 hash 值，只有计算 hash 值才能放入散列表.
2. 可以执行相等比较操作. hash 值可能会冲突，当 hash 值相同时，就要比较真正的 key 值.

== 函数比较简单，可以用与 < 函数类似的方式，通过重载操作符来实现：

```
bool operator==(const Point& a, const Point& b)
{
  return a.x == b.x; //自定义相等比较运算
}
```

散列函数可以用函数对象或 lambda 表达式实现，内部最好调用标准的 std::hash 函数对象，不要直接计算，否则很容易造成 hash 冲突

```
auto hasher = [](const auto& p)
{
  return std::hash<int>()(p.x);
}
```

有了相等函数和散列函数，自定义类型也可以放进无序容器了.

```
unordered_set<Point, decltype(hasher)> s(10, hasher);
s.emplace(7);
s.emplace(3);
```

如果只想要单纯的集合，字典，没有排序需求，就应该用无序容器，没有比较排序的成本，速度快.

只在末尾添加元素，就不用 deque/list，

只想快速查找元素，不用排序，就用 unordered_set.

Tips:

- 所有容器中，优先选择 array, vector，速度快，开销低.
- 多利用类型名，不要写死容器定义，因为容器的大部分接口相同，只要变动别名定义，就能随意改换不同容器.
- 有了 vector，不要再用 new/delete 来创建动态数组了.
- 标准库里有 stack, queue, priority_queue 三个容器适配器，它们并非真正的容器，而是由其它容器(通常是 deque, vector) 实现的，接口上有变化而内部结构相同.

标准容器 {
- 顺序容器 {
  - 线性表结构 连续存储 {
    - array 静态数组
    - vector 动态数组
    - deque 双端队列
  - 链式存储 {
    - list 双向队列
    - forward_list 单向链表.
- 有序容器 {
  - 红黑树结构 {
    - set/multi set 集合
    - map/multimap 关联数组/字典.
  - 二分查找效率高.
  - 插入元素有排序成本
  - 元素需要定义比较大小函数，即排序准则.
- 无序容器 {
  - 散列表结构 {
    - unordered_set/unordered_multiset 集合.
    - unordered_map/unordered_multimap 关联数组/字典.
  - 插入，查找成本低.
  - 元素需要定义散列函数和相等比较函数.
- 选择策略 {
  - 最优先选择的应该是 array 和 vector
  - 注重插入删除效率，可以选 list
  - 集合关系用 set，关联数组用 map
  - 单纯的集合，字典，不排序，用无序容器.
- 注意点，{
  - 存储元素是值语义.
  - 尽量为元素实现转移构造和转移赋值函数.
  - 存储裸指针要当心，可以考虑使用 shared_ptr