# Morden cpp notes — algorithm

Tuesday, December 8, 2020 2:15 PM

C++里的算法，指的是工作在容器上的一些泛型函数，会对容器内的元素实施各种操作，eg：
- remove 移除某个特定值
- sort 快速排序
- binary_search 执行二分查找
- make_heap 构造一个堆结构.

所有算法本质上都是 for 或者 while，通过循环遍历来逐个处理容器里的元素.

count算法，功能非常简单，统计某个元素出现次数，完全可以用 range-for 来实现同样的功能

```
vector<int> v = {1,2,3,1,2,3}
auto n1 = std::count(        count算法计算元素的数量
    begin(v), end(v), 1      begin(), end() 获取容器的范围.
);
int n2 = 0;
for (auto x : v) {           也可以用 for .
    if (x == 1) {
        n2++;
    }
}
```

不用for是为了追求更高层次的封装，抽象，是函数式编程的基本概念。而且有了可以就地定义函数的 lambda 表达式，算法形式和普通循环非常接近了。用算法+lambda，可以初步体验函数式编程，即函数套函数.

```
auto n = std::count_if(     // count_if 计算元素数量.
    begin(v), end(v),       // begin(), end() 获取容器的范围.
    [](auto x) {            // 定义一个 lambda 表达式.
        return x > 2;       // 判断条件.
    }
);
```

- iterator.

算法只能通过迭代器间接访问容器元素，是泛型编程的理念，与面向对象正好相反，分离了数据和操作，算法可以不关心容器的内部结构，以一致的方式操作元素，更加灵活.
缺点是对有的数据结构效率低。所以，对于merge, sort, unique等特别的算法，容器提供了替代成员函数.

容器一般会提供 begin(), end() 成员函数，调用它们就可以得到表示两个端点的迭代器，具体类型最好用auto推导.

```
vector<int> v = {1,2,3};
auto iter1 = v.begin();
auto iter2 = v.end();
```

建议使用更通用的全局函数 begin(), end(),

```
auto iter3 = std::begin(v);   //全局函数获取迭代器，自动类型推导.
```

常用的迭代器函数：
- distance() 计算两个迭代器间的距离.
- advance() 前进或后退N步.
- next()/prev() 计算迭代器前后的某个位置.

```
array<int, 5> arr = {0,1,2,3,4}     array静态数组容器.
auto b = begin(arr);    全局函数获取迭代器首端.
auto e = end(arr);      全局函数获取迭代器末端.
assert(distance(b,e) == 5);   迭代器距离
auto p = next(b);       获取下一个位置.
assert(distance(b,p) == 1);   迭代器的距离.
assert(distance(p,b) == -1);  反向计算迭代器的距离.
advance(p, 2);    迭代器前进2个位置，指向3.
assert(*p == 3);
asert(p == prev(e,2));   是末端迭代器的前两个位置.
```

手写循环的替代品：for_each.

```
vector<int> v = {1,2,3,4,5};
for(const auto& x : v){        range for 循环
    cout << x << ",";
}
auto print = [](const auto& x)     lambda 表达式.
{
    cout << x << ",";
};                                 常量迭代器
for_each(cbegin(v), cend(v), print);   for_each算法.

for_each(                         for_each算法，内部定义匿名
    cbegin(v), cend(v),           lambda表达式.
    [](const auto& x)
    {
        cout << x << ",";
    }
);
```

for_each 算法的价值，是把要做的事情分成了两部分，也就是两个函数：
1. 遍历容器元素.
2. 操纵容器元素.

所以 for_each 能够促使我们以函数式编程来思考，使用 lambda 封装逻辑，得到干净，安全的代码.

排序算法

```
auto print = [](const auto& x)
{
  cout << x << ",";
};
std::sort(begin(v), end(v));    快速排序
for_each(cbegin(v), cend(v), print);   for_each算法.
```

- 要求排序后保持元素的相对顺序，应用 stable_sort.
- 选出前几名 top N，应用 partial_sort
- 选出前几名，但不要求排出名次，best N，应用 nth_element.
- 中位数 median，百分位数 percentile，用 nth_element.
- 按某种规则把元素划分为两组，用 partition
- 第一名和最后一名，用 minmax_element.

下面展示了用法，注意"函数套函数"的形式：

```
//top 3
std::partial_sort(
    begin(v), next(begin(v), 3), end(v));

// best 3
std::nth_element(
    begin(v), next(begin(v), 3), end(v));

// median
auto mid_iter =
    next(begin(v), v.size()/2);
std::nth_element(begin(v), mid_iter, end(v));
cout << "median is " << *mid_iter << endl;

// partition, find all numbers larger than 9
auto pos = std::partition(
    begin(v), end(v),
    [](const auto& x)
    {
      return x > 9;
    }
);
for_each(begin(v), pos, print);  // print output

// find min and max.
auto value = std::minmax_element(
    cbegin(v), cend(v)
);
```

- 算法通常是随机访问迭代器，最好在 array/vector 上使用.
- 如果是 list 容器，应该调用函数 sort()，它对链表结构做了特别优化.
- 有序容器 set/map 已排序，可以直接对迭代器运算
- 无序容器不能排序，因为使用了散列表，元素无法交换位置.

查找算法.

binary_search 只能确定元素是否存在.

```
vector<int> v = {1,6,3,9,5};
std::sort(begin(v), end(v));     二分查找需要排序
auto found = binary_search(      //二分查找，只能确定元素是否存在.
    cbegin(v), cend(v), 3
);
```

想找到元素位置，需要用 lower_bound，它返回第一个大于或等于值的位置.

```
decltype(cend(v)) pos;   使用decltype声明一个迭代器.
pos = std::lower_bound(
    cbegin(v), cend(v), 3     找到第一个 >= 3 的位置.
);
found = (pos != cend(v)) && (*pos == 3);
assert(found);
pos = std::lower_bound(
    cbegin(v), cend(v), 7
);
found = (pos != cend(v)) && (*pos == 7);
assert(!found);
```

lower_bound 的返回值是一个迭代器，判断是否找到目标，条件有：
1. 迭代器是否有效.
2. 迭代器是不是要找的值.

upper_bound 返回第一个大于元素.

```
pos = std::upper_bound(
    cbegin(v), cend(v), 9
);
```

lower_bound, upper_bound 返回的是一个区间。区间往前是所有比被查值小的元素，往后是所有比被查值大的元素.

begin < x <= lower_bound < upper_bound < end

对于有序容器 set/map，可以用等价的成员函数 find/lower_bound/upper_bound.

```
multiset<int> s = {3,4,7,9,9,10};   //multiset 允许重复.
auto pos = s.find(7);
assert(pos != s.end());
auto lower_pos = s.lower_bound(7);   //获取区间的左端点.
auto upper_pos = s.upper_bound(7);   //获取区间的右端点.
```

查找区间的 find_first_of / find_end

```
vector<int> v = {1,9,11,3,5,7};
decltype(v.end()) pos;
pos = std::find(              找到第一个出现的位置.
    begin(v), end(v), 3
);
assert(pos != end(v));   与end()比较才能知道是否找到.

pos = std::find_if(
    begin(v), end(v),
    [](auto x) {              定义一个lambda表达式，判断是
      return x%2 == 0;        否为偶数.
};
assert(pos == end(v));

array<int, 2> arr = {3,5};
pos = std::find_first_of(
    begin(v), end(v),         查找一个子区间.
    begin(arr), end(arr)
);
assert(pos != end(v));
```

Tips
- 因为标准算法的名字实在太普通，一定要显式写出"std::"名字空间限定，避免无意的名字冲突.
- 容器还提供成员函数 rbegin()/rend()，用于逆序迭代 reverse，相应的也有全局函数 rbegin()/rend()，crbegin, crend().
- for_each 算法可以返回传入的函数对象，但因为 lambda 表达式只能被调用，没有状态，所以搭配 lambda 时，for_each 的返回值没有意义.
- equal_range 算法可以一次性获得 [lower_bound, upper_bound] 区间.
- C++14 允许算法并行处理，需要传入 std::execution::par 等策略参数.
- C++20 引用了 range，在名字空间 std::ranges 提供基于范围操作的算法，不必显式写出 begin(), end()，还重载了"|"实现管道操作.
- 在 lambda 表达式里可以用 return，可以结束循环，实现类似 for 里的 break.