

## 第十一章 标准模板库

# 目录

# 前言

## 学习目标

- 理解迭代器的工作原理和使用方法；
- 理解常见容器的特点并掌握它们的使用方法；
- 了解算法的类型并掌握常用调用对象的使用方法。

# 前言

标准模板库 (standard template library, STL) 是 C++ 标准库 (standard library) 的重要组成部分, 其包含以下几个部分:

- 容器 (container) 常用的数据结构, 包括 vector、list 等
- 算法 (algorithm) 操作容器的泛型算法, 包括查找、排序等
- 迭代器 (iterator) 处理不同类型容器的途径

# 迭代器

我们已经在 4.6.3 节介绍了迭代器 (iterator)，并使用它访问 vector 类型中的元素。本节我们考虑以下问题：

## 问题引入

给定一个 vector 或者数组以及一个数据值，要求查找给定的数据值是否在 vector 或者数组中，如果找到的话，返回该元素的地址；否则返回一个空指针。

# 迭代器

## 实现 Find 函数模板

根据问题要求，vector 类型和数组类型的查找算法函数模板如下：

### 查找函数模板

```
template<typename T>
const T* Find(const vector<T> &vec, const T &val) {
    for (int i=0; i<vec.size(); ++i)
        if (vec[i] == val)
            return &vec[i];
    return nullptr;
}

template<typename T>
const T* Find(const T *arr, int size, const T &val) {
    if (!arr || size <= 0)
        return nullptr;
    for (int i = 0; i<size; ++i)
        if (arr[i] == val)
            return &arr[i];
    return nullptr;
}
```

# 迭代器

实现 Find 函数模板

## 思考

观察上面两个函数模板，可以发现函数体中循环部分代码是一样的，那么是否可以通过只写一个版本的 Find 算法来处理 vector 和数组呢？

# 迭代器

实现 Find 函数模板

给定线性结构**第一个和最后一个元素的地址**，通过指针访问之，就可以实现一个通用算法如下：

## 通用算法

```
template<typename T>
const T* Find(const T *first, const T *last, const T &val) {
    if (!first || !last )
        return nullptr;
    for (;first!=last;++first)//last为尾后元素的地址
        if (*first == val)
            return first;
    return nullptr;
}
```



# 迭代器

实现 Find 函数模板

测试代码如下：

## 通用算法

```
if (auto p = Find(arr, arr + sizeof(arr) / sizeof(int), 5))  
    cout << *p << endl; //处理数组  
if (auto p = Find(&vi[0], &vi[vi.size() - 1]+1, 4))  
    cout << *p << endl; //处理vector
```

## 思考

以上代码需要用户自行指定第一个和尾后元素的地址，非常麻烦。那么我们是否可以对这两个元素进行封装以方便使用？

# 迭代器

实现 Find 函数模板

vector 的第一个元素和尾后元素取地址的操作包装为如下的 Begin 和 End 函数模板：

Begin 和 End 函数模板 (vector)

```
template<typename T>
const T* Begin(const vector<T> &vec) {
    return vec.size() > 0 ? &vec[0] : nullptr;
}
template<typename T>
const T* End(const vector<T> &vec) {
    return vec.size() > 0 ? &vec[vec.size()-1]+1 : nullptr;
}
```

# 迭代器

实现 Find 函数模板

数组首元素和尾后元素的封装类似：

## Begin 和 End 函数模板（数组）

```
template<typename T, size_t N>
const T* Begin(const T (&arr)[N]) {
    return arr;
}

template<typename T, size_t N>
const T* End(const T (&arr)[N]) {
    return arr + N;
}
```

其中，模板函数形参 arr 是实参数组的引用。

# 迭代器

实现 Find 函数模板

使用封装后的 Begin 和 End 调用 Find 函数的示例如下：

## 传入 Begin 和 End 调用 Find 函数

```
Find(Begin(vi), End(vi), 4);  
Find(Begin(arr), End(arr), 4);
```

和原始调用方式比较：

## 原始调用方式

```
Find(&vi[0], &vi[vi.size() - 1]+1, 4) //处理vector  
Find(arr, arr + sizeof(arr) / sizeof(int), 5) //处理数组
```

可见简洁很多。

# 迭代器

## 使用迭代器

每一种容器都有一个与之关联的迭代器。我们可以通过容器的成员函数 `begin` 和 `end` 获取第一个元素和尾后元素的迭代器，如：

### 使用迭代器

```
vector<int> vi = {0,1,2,3};  
vector<int>::iterator itb = vi.begin(); //itb指向vi 的首元素  
vector<int>::iterator ite = vi.end(); //ite指向vi的尾后元素
```

定义一个迭代器对象时，我们必须指明与之关联的容器和元素类型。如上述代码中迭代器指向 `vector<int>` 中的元素。

通常我们会用 `auto` 来简化迭代器定义

### 利用 `auto` 简化迭代器定义

```
auto itb = vi.begin(); //利用auto简化定义
```

# 迭代器

## 使用迭代器

无论指向何种容器，迭代器都支持以下操作：

- **解引用与成员选择**：`*iter`, `iter->member`（等价于 `(*iter).member`）；
- **自增运算符**：`++iter`, `iter++`；
- **赋值运算符**：`iter1 = iter2`；
- **关系 `==` 和 `!=` 运算**：`iter1 == iter2`, `iter1 != iter2`。

通常，我们使用迭代器来遍历容器中的元素

### 使用迭代器进行遍历

```
for(auto it = vi.begin(); it != vi.end(); ++it){//遍历vector  
    cout << *it <<endl;  
}
```

几乎 STL 提供的所有算法都是通过迭代器实现对容器中元素的操作，即通过接受由 `begin` 和 `end` 划定的左闭合区间 `[begin,end)`，对区间内元素进行操作。

# 迭代器

## 使用迭代器

迭代器的简单分类如下：

- **输入 (input) 迭代器**：只能单步向前迭代（自增运算 ++），**不允许修改由该类迭代器引用的元素**；
- **输出 (output) 迭代器**：该类迭代器和输入迭代器相似，也只能单步向前迭代，不同的是该类迭代器对引用的元素**只能执行写操作**；
- **前向 (forward) 迭代器**：该类迭代器可以在一个正确的区间中进行读写操作，它拥有输入和输出迭代器的特性，**仅支持自增运算**；
- **双向 (bidirectional) 迭代器**：该类迭代器是在前向迭代器基础上提供了单步向后迭代的功能，**支持自增 (++) 和自减 (-) 运算**；
- **随机访问 (random access) 迭代器**：该类迭代器具有上面所有迭代器的功能，并能**直接访问容器中任意一个元素**，支持  $iter+n$ ,  $iter-n$ ,  $iter+=n$ ,  $iter-=n$ ,  $iter1-iter2$ 。

### 注意

一个迭代器的类型取决于**与其关联的容器类型**，比如一个指向 `vector` 类型的迭代器的类型为随机访问迭代器。

STL 中的容器主要由两大类组成：

- 顺序容器
- 关联容器

大多数容器都支持以下操作：

- 关系运算
- 赋值运算成员
- begin 和 end 成员
- empty 成员
- size 成员
- clear 成员



# 容器

## 容器概述——创建容器

一般而言，每个容器都定义在一个与容器名相同名称的头文件中，使用时包含之即可

### 包含容器头文件

```
#include <vector>
```

每一种容器均被定义为类模板，因此在使用时需要提供额外的模板参数信息

### 提供容器模板参数

```
vector<string> vs;
```

# 容器

## 容器概述——cbegin 和 cend

C++11 还为每一种容器提供了 `cbegin` 和 `cend` 成员，分别返回第一个元素和尾后元素的 `const` 迭代器，不允许对指向的元素执行写操作

### 包含容器头文件

```
vector<int> vec = {0, 1, 2, 3, 4};  
auto it1 = vec.begin(); //返回第一个元素的迭代器  
auto it2 = vec.cbegin(); //返回第一个元素的const迭代器  
*it1 = 4; //正确：修改第一个元素的值  
*it2 = 5; //错误：it2为const 迭代器，不允许修改指向的对象
```

对于 `begin` 成员，只有当容器是 `const` 类型，才返回 `const` 类型迭代器；否则返回非 `const`。为了避免不必要的修改错误，C++ 增加了上述 `cbegin` 和 `cend` 成员，即无论容器是否为 `const`，都返回 `const` 迭代器。

# 容器

## 容器概述——cbegin 和 cend

C++11 还为每一种容器提供了 `cbegin` 和 `cend` 成员，分别返回第一个元素和尾后元素的 `const` 迭代器，不允许对指向的元素执行写操作

### 包含容器头文件

```
vector<int> vec = {0, 1, 2, 3, 4};  
auto it1 = vec.begin(); //返回第一个元素的迭代器  
auto it2 = vec.cbegin(); //返回第一个元素的const迭代器  
*it1 = 4; //正确：修改第一个元素的值  
*it2 = 5; //错误：it2为const 迭代器，不允许修改指向的对象
```

对于 `begin` 成员，只有当容器是 `const` 类型，才返回 `const` 类型迭代器；否则返回非 `const`。为了避免不必要的修改错误，C++ 增加了上述 `cbegin` 和 `cend` 成员，即无论容器是否为 `const`，都返回 `const` 迭代器。

# 容器

## 容器概述——插入和删除元素

以下代码展示 vector 的插入和删除数据操作：

### vector 的插入和删除数据操作

```
vector<int> vec = {0, 1, 2, 3, 4};  
vec.insert(vec.begin(), 10);  
vec.erase(vec.begin()+1);
```

### 说明

- 除 C++11 新增的 array 容器以外，其它容器都是可变长的

# 容器

## 容器概述——插入和删除元素

C++11 为可变长容器新增的 `emplace` 成员，示例如下：

### 定义一个 `foo` 类

```
struct Foo{
    Foo(const string &name, int id) :m_name(name),
        m_id(id) {}
    string m_name;
    int m_id;
};
```

### `emplace` 与 `insert` 成员用法区别

```
vector<Foo> vf;
vf.push_back(Foo("Lisha", 12));
vf.insert(vf.begin(), Foo("Mandy", 13));
vf.emplace_back("Kevin", 11);
vf.emplace(vf.begin(), "Rosietta", 10);
```

### 说明

- `emplace` 成员通过一个参数包接受的参数来构造一个元素并将之插入容器中。
- `emplace_back` 函数调用把两个实参传递给 `Foo` 类的构造函数，并在 `vf` 的末尾利用这两个参数值构造一个新元素。
- 与 `emplace_back` 成员相比，`push_back` 和 `insert` 成员只能将已构造的元素移动或复制到容器中相应的位置，不能在指定位置直接构造一个元素。

# 容器

## 容器概述——swap 操作

swap 操作交换两个相同类型容器的数据：

### swap 操作

```
vector<int> v1 = {0, 1}; //2个元素的vector  
vector<int> v2 = {0, 1, 2, 3}; //4个元素的vector  
swap(v1, v2);
```

### 说明

- 调用 swap 函数之后，v1 将包含 4 个元素，v2 将包含 2 个元素。

### 注意

- 除 array 之外，swap 函数不会执行任何数据复制、插入或删除操作。
- 对于 array 来说，swap 会真正交换相同位置的元素。

### 思考

对于 array 和其他容器而言，swap 操作之后与容器绑定的迭代器、指针以及所指向的元素是否发生了变化？各发生了怎样的变化？

# 容器

## 顺序容器

### 顺序容器

顺序容器都是线性结构，提供了元素的快速顺序访问能力。

### 注意

但对于非线性访问和元素增减操作，它们有很大的性能差别。

- 与 `vector`、`string`、`deque`、`array` 等容器绑定的迭代器支持随机访问
- 与 `list` 绑定的迭代器支持双向单步迭代
- 与 `forward_list` 绑定的迭代器只支持前向单步迭代。

# 容器

顺序容器——使用 array

定长数组容器 array 的使用如下：

## 使用 array

```
array<int, 4> arr = {1,2,3,4};  
for (auto it = arr.begin(); it != arr.end(); ++it)  
    cout << *it << endl;  
array<int, 4> arr2 = arr; //array对象允许复制  
arr2.fill(0); //所有元素赋值为0
```

## 说明

- 相比于普通数组，array 更安全、更易使用
- 和普通数组的行为类似，array 不支持插入、删除等改变容器大小的操作
- 和普通数组相比，array 对象支持赋值和复制操作，还能通过 size 成员获取数组的大小。



# 容器

## 顺序容器——使用 deque

双端队列容器 deque 的使用如下：

### 使用 deque

```
deque<int> dq = {1,2,3};  
dq.push_back(4); //尾部插入一个元素  
dq.push_front(0); //首部插入一个元素  
cout << dq[3]<< endl; //随机访问
```

### 说明

- 与 vector 类似，deque 支持随机访问。
- 与 vector 不同的是，deque 可以在首尾两端进行快速地插入和删除操作

### 注意

- 虽然 deque 也支持随机访问操作，但它的访问效率比容器要低很多。因为 deque 由一些在内存中互相独立的动态数组组成

# 容器

## 顺序容器——使用 forward\_list

forward\_list 的使用如下：

### 使用 forward\_list

```
forward_list<int> flst = { 2, 3 };  
flst.push_front(1); //在flst首部插入数据  
flst.insert_after(flst.before_begin(), 0); //同上  
for (auto it = flst.begin(); it != flst.end(); ++it)  
    cout << *it << " "; //打印输出: 0 1 2 3
```

要获取 forward\_list 中的元素数目，可以使用 distance 函数：

### 使用 distance 获取 forward\_list 元素数目

```
cout << "size: " << distance(flst.begin(), flst.end()  
    ()) << endl;
```

### 说明

- forward\_list 无法访问到给定位的前驱，仅提供给定位后的插入删除操作
- before\_begin 成员返回的迭代器是不能解引用的
- 出于性能考虑，forward\_list 放弃了 size 函数

### 注意

- 与 vector、array、deque 相比，forward\_list 不支持随机访问。但对于元素的插入、删除、移动等操作，它的性能要好于前三者

# 容器

## 顺序容器——使用 list

list 的使用如下：

### 使用 list

```
list<int> lst1 = {2,3}, lst2 = {1};  
lst1.push_back(5); //在lst1的尾部插入元素5  
lst2.push_front(0); //在lst2的首部插入元素0  
auto pos = find(lst1.begin(),lst1.end(),5); //找到指向  
元素5的迭代器  
lst1.insert(pos, 4); //在此位置插入元素4  
lst1.splice(lst1.begin(), lst2); //将lst2插入到lst1中  
第1个元素位置  
for (auto it = lst1.begin(); it != lst1.end(); ++it)  
cout << *it << " "; //打印输出: 0 1 2 3 4 5
```

### 说明

- list 是一个双向链表
- 除了 insert、push\_back 等成员可以执行插入操作外，可以使用 splice 成员将一个 list 中的元素转移到另外一个 list 中
- splice 函数调用执行完后，lst2 变为空列表

# 容器

## 顺序容器——使用 list

list 的使用如下：

### 使用 splice 移动某一范围元素

```
list<int> lst3 = { 6,7,8 };  
auto it = lst3.begin();  
advance(it, 2); //将it后移两个位置  
lst1.splice(lst1.end(), lst3, lst3.begin(),it); //将  
    lst3中前两个元素转移到lst1的尾部
```

### 说明

- splice 调用将 lst3 中从 begin 到 it 范围内的元素 (包括 begin 但不包括 it) 转移到 lst1 的尾部
- 上面 splice 调用结束时, lst1 的尾部新增两个元素, lst3 剩余 1 个元素。

# 容器

## 顺序容器——顺序容器的选择

没有性能完美的容器，在选择顺序容器时，我们需要考虑以下几点：

- 如果需要高效的随机存取，不在乎插入和删除的效率，则使用 `vector`；
- 如果需要大量的插入和删除元素，不关心随机存取的效率，则使用 `list`；
- 如果需要随机存取，并且关心两端数据的插入和删除效率，则使用 `deque`；
- 如果仅在读取输入的数据时在容器的中间位置插入元素，数据输入完毕之后仅需要随机访问，则可考虑在输入时将元素读入到一个 `list` 容器中，然后对此容器使用 `sort` 函数排序，最后将排序后的 `list` 复制到一个 `vector` 容器中。
- 如果程序既需要随机访问又必须在容器的中间位置插入或删除元素，那么我们需要比较随机访问 `list` 和在 `vector` 中间插入或删除元素时移动元素的代价。

### 注意

如果无法确定某种应用应该采用哪种容器，则编写代码时尝试只使用 `vector` 和 `list` 容器都提供的操作，方便以后进行转换。

# 容器

## 关联容器

### 关联容器

不同于顺序容器，关联容器采用非线性结构。通常情况下，关联容器是通过树结构实现的。关联容器中的元素通过关键字来访问，而顺序容器中的元素通过元素的位置来访问。STL 中两个主要的关联容器是set和map。

# 容器

关联容器——使用 set

set

set 中每个元素只包含一个关键字，与数学上的集合类似，set 不包含重复的元素，且它们都是有序的。

## 例 11.1

统计输入的一组数字中不同数字的个数，并将它们排序输出

# 容器

## 关联容器——使用 set

### set

set 中每个元素只包含一个关键字，与数学上的集合类似，set 不包含重复的元素，且它们都是有序的。

### 例 11.1

统计输入的一组数字中不同数字的个数，并将它们排序输出

### 例 11.1

```
set<int> counter; //创建一个关键字类型为int的空set对象
int number;
while (cin>>number) //输入数字
counter.insert(number); //将输入的数字插入到set中
cout << "不同的数字的个数: " << counter.size() <<
endl; //获取元素个数
for (auto &i : counter) //遍历每个元素
cout << i << " "; //输出每个元素
```

### 说明

- 向 set 插入元素时，如果 set 中已有此元素，则将其抛弃；否则，按序将其插入。
- 输入：1 8 4 2 0 1 4 3 5 4，  
输出：不同数字的个数：7  
0 1 2 3 4 5 8
- 可见重复元素被排除，剩余按升序排列



# 容器

## 关联容器——使用 set

利用 find 函数查找、利用 erase 删除 set 中元素：

### find 查找 set 中元素

```
vector<int> v = { 1, 8, 4, 2, 0, 1, 4, 3, 5, 4, 7 };  
set<int> s(v.begin(), v.end()); //利用指向vector的迭代器范围创建set  
auto it = s.find(0); //查找关键字为0的元素
```

### erase 删除 set 中元素

```
s.erase(it); //删除关键字为0的元素  
s.erase(s.find(3), s.find(7)); //删除[3,7)范围内的元素  
for (auto &i : s)  
cout << i << " "; //打印输出: 1 2 7 8
```

### 说明

- find 查找时，待查元素如果存在，则返回该元素的迭代器；否则返回尾后迭代器 (end)。
- erase 成员的迭代器范围参数为左闭合区间，即从 begin 开始，于 end 之前结束。

### 注意

- 查找 set 中元素时亦可使用全局 find 函数，但成员 find 函数的性能更好。
- 调用 erase 成员不影响与 set 中其它元素绑定的迭代器

# 容器

## 关联容器——使用 map

介绍 map 之前，首先了解一种标准库类型模板 pair：

### 使用 pair

```
pair<int, int> p1; //保存两个int类型数据
pair<string, int> p2 = {"Hello", 0}; //列表初始化两个成员
auto p3 = make_pair("Hello", 1); //make_pair函数返回一个pair对象
cout << p2.first << p2.second << endl; //访问pair中数据成员
```

### pair

pair 定义在头文件 `utility` 中，包含两部分数据成员。

### 说明

- pair 的两个数据成员是公有的，名字分别为 `first` 和 `second`，可以直接访问

### erase 删除 set 中元素

```
s.erase(it); //删除关键字为0的元素
s.erase(s.find(3), s.find(7)); //删除[3,7)范围内的元素
for (auto &i : s)
    cout << i << " "; //打印输出: 1 2 7 8
```

# 容器

关联容器——使用 map

使用 map:

## 例 11.2

统计输入的一组数字中每个数字出现的次数

## 例 11.2

```
map<int, int> counter; //创建一个存放pair<int,int>类  
    型的map对象  
int number;  
while (cin >> number)  
    ++counter[number];  
for (auto &i : counter) //遍历map 中每个元素  
    cout << i.first << ": " << i.second << endl; //打印关  
    键字和值
```

输入: 1 2 4 4 5 3 2 4 7 0 2

输出:

0: 1

1: 1

2: 3

...

## pair

map 与 set 类似，都是有序容器。但 map 中的元素是 pair 类型，第一个成员为用于索引的关键字，第二个成员为与关键字相关的值。

## 说明

- map 提供了下标运算，用来获取与关键字关联的值
- 执行第四行代码时，如找到关键字为 number 的元素则其值自增，否则以此关键字生成新的元素，本例中值初始化默认为 0

# 容器

## 关联容器——使用 map

### 使用 map:

#### 例 11.2

统计输入的一组数字中**每个数字出现的次数**

#### 例 11.2

```
map<int, int> counter; //创建一个存放pair<int,int>类
                        型的map对象
int number;
while (cin >> number)
    ++counter[number];
for (auto &i : counter) //遍历map 中每个元素
    cout << i.first << ": " << i.second << endl; //打印关
                        键字和值
```

输入: 1 2 4 4 5 3 2 4 7 0 2

输出:

0: 1

1: 1

2: 3

...

#### pair

map 与 set 类似，都是有序容器。但 map 中的元素是 pair 类型，第一个成员为用于索引的关键字，第二个成员为与关键字相关的值。

#### 注意

- map 提供了下标运算，用来获取与关键字关联的值
- 执行上面第四行代码时，如找到关键字为 number 的元素则此关键字的值自增，否则以此关键字生成新的元素，本例中值初始化默认为 0

# 容器

关联容器——使用 map

使用 map:

## 使用 insert 成员添加元素

```
counter.insert({ 3, 0 }); //C++11新特性  
counter.insert(make_pair(3, 0));
```

## 检测插入是否成功

```
auto res=counter.insert(pair<int, int>(2,0));//自动推  
导res的类型  
if (!res.second) //关键字2已经存在  
++res.first->second; //关键字为2的元素的值自增
```

## 说明

- insert 函数返回一个 pair 对象，该对象的第一个成员为一个指向 map 中给定关键字的迭代器，第二个成员是一个 bool 值。给定关键字已存在则其值为 false；否则为 true

# 容器

关联容器——使用 map

提示：C++11 新标准允许为关联容器进行列表初始化

## 关联容器列表初始化

```
set<string> names = {"Kevin", "Lisha", "Mandy", "Rosieta"};  
map<string, unsigned long long> contact = {  
{"Kevin", 15387120503}, {"Rosieta", 15387120506}};
```

## 说明

- 对于 set，每个元素的类型即为关键字类型。对于 map，每个元素的类型为一对花括号括起来的 pair 类型

# 容器

关联容器——使用 map

提示：C++11 新标准允许为关联容器进行列表初始化

## 关联容器列表初始化

```
set<string> names = {"Kevin", "Lisha", "Mandy", "Rosieta"};  
map<string, unsigned long long> contact = {  
{"Kevin", 15387120503}, {"Rosieta", 15387120506}};
```

## 说明

- 对于 set，每个元素的类型即为关键字类型。对于 map，每个元素的类型为一对花括号括起来的 pair 类型

# 容器

## 关联容器——使用 multimap

multimap 的使用范例：

### 例

set 和 map 中的关键字必须是唯一的。但有些情况下，比如我们存放电话簿时，同一个人可能有不同的手机号码，这时候应该怎么办？

### 使用 multimap

```
multimap<string, unsigned long long> contact;  
contact.insert({ "Kevin", 15387120503 });  
contact.insert({ "Kevin", 15387120506 });  
for (auto &i : contact)  
    cout << i.first << ": " << i.second << endl;  
auto entries = contact.count("Kevin");  
auto it = contact.find("Kevin");  
while (entries) {  
    cout << it->second << endl; //打印电话号码  
    ++it; //移动到下一个记录  
    --entries; //计数器自减  
}
```

### 说明

- find 语句将返回第一个关键字为 Kevin 的元素的迭代器
- 利用 count 返回的值，不断递增迭代器 it，直到 Kevin 的所有号码被打印。



# 容器

## 高效使用容器

当我们利用容器实现某种功能时，可能会面临多种容器的选择。如果选择的容器不合理的话，将会大大**降低程序的效率**。本节将介绍一些常用的容器使用原则。

### 使用 `empty` 检查容器是否为空

`empty` 和 `size` 成员均能用于检测容器是否为空，但是 `empty` 存在两个优势：

- `empty` 总能保证常数时间内返回
- `empty` 是所有容器通用的操作

而成员 `size` 不总是通用的，比如 `forforward_list` 就没有提供 `size` 成员。

因此建议使用成员 `empty` 来检查容器是否为空。

# 容器

## 高效使用容器

### 使用存放指针的容器

如果容器中存放的对象是大对象（占用较大内存空间），那么在操作容器过程中复制大对象会使得程序付出很大的性能代价。此时我们可以考虑使用指针的容器而不是对象的容器。

比如我们有以下“大”对象类

# 容器

## 高效使用容器

### 定义“大”对象类

```
struct LargeData {  
    LargeData(int id): m_id(id){}  
    int m_id;  
    int m_arr[1000];  
};
```

### reverse “大” 对象

```
vector<LargeData*> vp;  
vector<LargeData> vo;  
for (int i = 0; i < 50000; i++){  
    int n = rand()%10000000; //生成一个随机数  
    vp.emplace_back(new LargeData(n)); //尾插一个元素  
    vo.emplace_back(n); //尾插一个元素  
}  
reverse(vo.begin(), vo.end()); //翻转对象  
reverse(vp.begin(), vp.end()); //翻转指针
```

### 说明

- 对 vo 的 reverse 操作会使用对象的复制操作，但对 vp 的 reverse 操作只涉及到指针复制，没有额外类对象的复制行为。
- 因此，对 vp 执行 reverse 操作花费的时间会比对 vo 执行 reverse 操作花费的时间小的很多。

# 容器

## 高效使用容器

同时，为了方便地维护内存，避免内存泄漏问题，我们可以在容器中存放 `unique_ptr`

### 在容器中存放 `unique_ptr`

```
vector<unique_ptr<LargeData>> vsp;
```

### 向 `vector` 内插入 `unique_ptr` 对象

```
vp.push_back(make_unique<LargeData>(1)); //  
    make_unique为C++14 标准  
vp.push_back(std::move(unique_ptr<LargeData>(new  
    LargeData(2))));  
vp.emplace_back(new LargeData(3));
```

### 说明

- 第一条语句中调用 C++14 标准下的 `make_unique` 函数
- 第二条语句使用 `move` 函数将一个临时对象移动到 `vp` 的尾部
- 第三条语句使用容器类成员 `emplace` 函数在 `vp` 的尾部直接构造一个对象。

### 注意

对指针容器使用排序算法时，我们需要定义基于对象的比较函数。如果使用 STL 的默认的比较函数，其结果是基于指针大小的比较，而不是对象的比较。

# 容器

## 高效使用容器

### 使用算法和区间成员

相比单元素遍历操作，使用区间成员的优势在于:1) 更少的函数调用；2) 更少的元素移动；3) 更少的内存分配

#### 插入元素：使用单元素遍历操作

```
int arr[] = { 1,2,4,10,5,4,1,8,20,30,15 };  
vector<int> vi;  
for (int i = 0; i < 7; i++)  
    vi.push_back(arr[i]);
```

#### 插入元素：使用区间成员

```
vi.assign(arr, arr + 7);
```

#### 说明

- 上面代码当每次 vi 的容量 (capacity) 小于实际需求时，vector 会自动分配更大的存储空间，将已有元素移动到新的空间中，然后添加新元素
- 下面的代码得到简化，并且由于不需要预先分配空间，减少了内存分配和数据移动的操作，提高了性能

# 容器

## 高效使用容器

### 使用算法和区间成员

相比单元素遍历操作，使用区间成员的优势在于:1) 更少的函数调用；2) 更少的元素移动；3) 更少的内存分配

#### 插入元素：使用单元素遍历操作

```
int arr[] = { 1,2,4,10,5,4,1,8,20,30,15 };  
vector<int> vi;  
for (int i = 0; i < 7; i++)  
    vi.push_back(arr[i]);
```

#### 注意

对于 vector 容器，成员 **capacity** 指的是当前状态下，容器**能容纳**的元素数目，而 **size** 指的是当前容器中**实际**的元素数目。

#### 插入元素：使用区间成员

```
vi.assign(arr, arr + 7);
```

# 容器

## 高效使用容器

### 使用 reserve 成员

对于 vector 容器，如果预先知道数据需要的空间大小，可以利用 reserve 成员预先分配空间，这样会避免重新分配空间和移动已有元素产生的代价。

### 插入元素：使用单元素遍历操作

```
vector<int> vi;  
cout << "预留前，容量：" << vi.capacity() << "大小：" << vi  
    .size() << endl;  
vi.reserve(1000);  
cout << "预留后，容量：" << vi.capacity() << "大小：" << vi  
    .size() << endl;
```

### 注意

使用 reserve 只是重新分配内存空间，改变它的容量，但不会对 vector 产生 resize 行为，因此容器中的内容是不变的

### 运行结果

输出：预留前，容量：0，大小：0

输出：预留后，容量：1000，大小：0

# 容器

## 高效使用容器

### 使用有序的 vector 容器

如果我们的操作是分阶段的，如一系列插入操作-> 查询操作  
那么我们可以：

- ① 使用有序关联容器完成插入
- ② 使用关联容器创建有序 vector
- ③ 使用 vector 进行查询

### 使用有序的 vector 容器

```
multiset<int> s; //利用multiset存放有序元素
int number;
while (cin >> number) //插入元素
    s.insert(number);
vector<int> v(s.begin(), s.end()); //创建有序vector
if (binary_search(v.begin(), v.end(), 10)) //二分查找
    元素10
    cout << "10 is found" << endl;
else cout << "10 is not found" << endl;
```

### 运行结果

输入：10 20 10 30 15 20 10  
输出：10 is found



# 容器

## 高效使用容器

### 正确使用 map 的 insert 和下标运算符

对于 map 来说，其成员 insert 和下标运算符有着不同的功能。

- 使用下标运算符意味着可能插入新的元素或覆盖已有元素的值
- insert 专用于插入，不会覆盖已有元素
- at 成员则只对元素进行访问

如果你需要修改已有元素的值且不在意下标运算符是否会插入新的元素，则可以尽情享受下标运算符带来的方便。比如例 11.2：

#### 前例 11.2 统计输入的一组数字中每个数字出现的次数

```
map<int, int> counter;  
int number;  
while (cin >> number)  
    ++counter[number];  
for (auto &i : counter)  
    cout << i.first << ": " << i.second << endl;
```

#### 运行结果

```
输入: 1 2 4 4 5 3 2 4 7 0 2  
输出:  
0: 1  
1: 1  
2: 3  
...
```

# 容器

## 高效使用容器

### 使用成员函数代替同名的算法

有些容器的成员函数名和 STL 中算法的名字相同，它们都实现某种特定的功能。通常情况下，**成员函数的效率要好于全局算法**。

#### 前例 11.2 统计输入的一组数字中每个数字出现的次数

```
vector<int> v = { 3, 7, 3, 11, 3, 3, 2 };  
set<int> s(v.begin(), v.end());  
auto it1 = find(s.begin(), s.end(), 10); //查找速度慢  
auto it2 = s.find(10); //查找速度快
```

#### 说明

- 使用 find 函数查找时为依次比较，为线性复杂度
- 使用 set 成员 find 查找时会利用 set 的有序性快速查找，为对数时间复杂度

# 泛型算法

标准库提供了很多泛型算法，这些算法可以用于不同类型的容器。它们有一致的结构，大多数算法都接受一个范围迭代器，对此范围内的元素进行处理。处理的元素放在什么样的容器中，算法并不需要了解。本节我们将介绍算法的框架和使用方法，算法详细介绍可参考在线手册：<http://zh.cppreference.com/>。

# 泛型算法

## 算法概述

### 标准库算法

依据算法对元素的访问方式，标准库算法主要有三大类：

- 只读型
- 写入型
- 重排型

大多数算法都定义在头文件 `algorithm` 中，基本语法格式见书中附录 C。

# 泛型算法

## 算法概述

### 只读型算法

只是读取迭代器范围内的元素，不会改变元素的内容

#### 只读算法案例 1——find

```
vector<int> v = { 3, 7, 3, 11, 3, 3, 2 };  
auto it = find(v.begin(), v.end(), 10);  
cout << "10 is " << (it != v.end() ? "found" : "not  
found");
```

#### find 函数

用来遍历给定范围内的元素  
是否存在一个特定值

#### 说明

- 前两个参数为迭代器范围，第三个参数为待搜索值
- 它从开始位置依次将每个元素与给定值比较
- 如找到，返回第一个与给定值相等的元素的迭代器
- 否则返回第二个参数表示搜索失败

# 泛型算法

## 算法概述

### 只读算法案例 2——accumulate

```
int sum = accumulate(v.begin(), v.end(), 0);
```

### 只读算法案例 2——accumulate

```
vector<string> vs = { "Hello ", "world" };  
string s1 = accumulate(vs.begin(), vs.end(), string  
    ()); //正确  
string s2 = accumulate(vs.begin(), vs.end(), ""); //  
    错误
```

### accumulate 函数

计算特定范围内元素的和

### 说明

- 前两个参数表示元素范围的迭代器，第三个参数表示和的初始值
- 初始值决定了返回值的类型
- 下方代码块第三条语句中的第三个参数类型为 `const char*`。由于 `const char*` 没有定义 `+` 运算，因此产生编译错误

# 泛型算法

## 算法概述

### 只读算法案例 2——accumulate

```
int sum = accumulate(v.begin(), v.end(), 0);
```

### 只读算法案例 2——accumulate

```
vector<string> vs = { "Hello ", "world" };  
string s1 = accumulate(vs.begin(), vs.end(), string  
    ()); //正确  
string s2 = accumulate(vs.begin(), vs.end(), ""); //  
    错误
```

### accumulate 函数

计算特定范围内元素的和

### 注意

如果元素的类型为 string，那么将范围内所有的 string 连接起来，其中第三个参数类型必须是 string 类型，不能是字符串常量

# 泛型算法

## 算法概述

### 写入型算法

将元素的值写入到容器中

#### 写入型算法案例——fill

```
vector<int> v1(10),v2(15);  
fill(v1.begin(), v1.end(), 1); //将容器v1中所有元素重  
置为1  
copy(v1.begin(), v1.end(), v2.begin()); //将v1中的元  
素复制到v2中
```

#### fill 函数

将给定的值写入到指定的范围内

#### copy 函数

将给定范围内的元素依次复制到第三个迭代器参数指定的起始位置

### 说明

- fill 函数前两个参数为迭代器范围（目的序列），第三个参数为写入值
- copy 函数前两个迭代器参数表示输入范围，第三个迭代器表示目的序列的起始位置。



# 泛型算法

## 算法概述

### 重排型算法

重新排列容器中的元素顺序

#### 重排型算法案例 1——sort

```
vector<int> v = { 3, 7, 3, 11, 3, 3, 2 };  
sort(v.begin(), v.end()); //升序排序
```

#### sort 函数

使输入序列中的元素有序，默认的元素比较方式为 < 运算符

#### 重排型算法案例 2——stable\_sort

```
stable_sort(v.begin(), v.end());
```

#### stable\_sort 函数

与 sort 函数作用相同，但会保持相等元素的相对位置

#### 注意

数值相等的元素的相对位置  
在使用 sort 排序后可能会改变

# 泛型算法

## 算法概述

### 注意

如果容器元素是用户自定义类型，则需要提供元素的 `<` 运算符

### 重排型算法案例 3——自定义 `<` 运算符

```
struct LargeData {  
    bool operator< (const LargeData rhs) {  
        return m_id < rhs.m_id; //比较对象的id  
    } //其它成员与11.2.4节相同  
};
```

### 说明

- 自定义类型 `LargeData` 类有了定义的 `<` 运算符之后，方可使用 `sort` 进行排序

### 重排型算法案例 3——使用自定义 `<` 运算符 `sort`

```
vector<LargeData> vo;  
for (int i = 0; i < 50000; i++)  
    vo.emplace_back(rand() % 10000000);  
sort(vo.begin(), vo.end()); //按元素id的升序排序
```

# 泛型算法

## 向算法传递函数

为了提高程序的效率，我们在 11.2.4 节中建议使用指针容器代替对象容器：

### 指针容器代替对象容器

```
vector<LargeData*> vp;
```

由于 sort 算法将会按照**指针大小排序**而不会对**指针指向的对象**进行排序，所以我们需要**定义自己的比较方式**。sort 算法的第二个版本的第三个参数接收一个**二元谓词**，即我们所定义的比较方式。

### 谓词

谓词是一个可以调用的表达式，其返回的结果能用于条件测试。

标准库算法使用的谓词有两种：

- 一元谓词 (unary predicate)
- 二元谓词 (binary predicate)

一元谓词只接受一个参数，二元谓词有两个参数。

下面我们介绍向算法传递可调用对象的三种方式：使用函数、使用函数对象和使用 lambda 表达式

# 泛型算法

## 算法概述——使用函数

我们定义下面一个函数，该函数接受两个 `LargeData` 类型指针，比较的是指针指向的对象的 `id`

### 向算法传递函数示例 1——使用函数（定义函数）

```
bool Less(const LargeData *a, const LargeData *b) {  
    return a->m_id < b->m_id;  
}
```

### 向算法传递函数示例 1——使用函数（传递函数）

```
sort(vp.begin(), vp.end(), Less);
```

### 说明

- 左面代码执行完以后，`vp` 中的元素将会按照 `id` 升序排列
- 当 `sort` 算法需要比较两个元素时，便会调用 `Less` 函数

# 泛型算法

## 算法概述——使用函数对象

除了向算法传递普通的函数，我们还可以传递一个函数对象

### 向算法传递函数示例 2——使用函数对象（定义）

```
struct Compare {  
    bool operator()(const LargeData *a, const  
        LargeData *b) {  
        return a->m_id < b->m_id;  
    }  
};
```

### 向算法传递函数示例 2——使用函数对象（传递）

```
sort(vp.begin(), vp.end(), Compare());
```

### 说明

- 上方代码为 Compare 类定义一个函数调用运算符，其参与前述 Less 函数一样，功能也是比较与两个形参绑定的 LargeData 对象的 id
- 下方 sort 函数调用中的第三个实参为通过 Compare 默认构造函数创建的一个函数对象
- 函数对象可以保存调用时的状态，相比于普通函数更为灵活

# 泛型算法

## 算法概述——使用函数对象

函数对象可以保存调用时的状态。相比于普通函数，函数对象更加灵活，能够完成函数不能完成的任务。下面我们通过 Checker 类和 find\_if 算法来查找容器中第 n 个元素：

### 向算法传递函数示例 3——查找第 n 个元素

```
struct Checker{  
    int m_cnt = 0, m_nth;  
    Checker(int n) :m_nth(n) {}//初始化设定值  
    bool operator()(int) { return ++m_cnt == m_nth; }  
};
```

### 向算法传递函数示例 3——查找第 n 个元素

```
vector<int> v = { 3, 7, 3, 11, 3, 3, 2 };  
auto i=find_if(v.begin(),v.end(),Checker(4));//返回第  
4个元素的迭代器
```

### 说明

- Checker 类的两个数据成员分别用来计数 (m\_cnt) 和保存设定值 (m\_nth)
- 每一次调用 Checker 对象，该对象的计数器 (m\_cnt) 就会自增，当计数器增加到设置值时，该调用返回真
- 当调用返回真时，find\_if 返回指向当前元素的迭代器

# 泛型算法

## 算法概述——使用 lambda 表达式

一个 lambda 表达式为一个**可调用的代码单元**（见 5.6.2 节，第 116 页），因此我们也可以向算法传递一个 lambda 表达式，例如：

### 向算法传递函数示例 4——使用 lambda 表达式

```
sort(vp.begin(), vp.end(), [](const LargeData *a,  
    const LargeData *b) {return a->m_id < b->m_id;});
```

### 说明

- lambda 表达式的捕获列表为空
- 函数形参为两个指针类型
- 函数体与 Less 函数和 Compare 函数调用运算符一样，都是比较两个指针指向的对象的 id

和上面两种方法相比，使用 lambda 表达式：

- 它不需要额外定义一个函数或一个函数对象类
- 可以利用捕获列表访问外围对象

如果传递的可调用对象的**操作比较简单**且**只在局部使用**，lambda 表达式是最佳选择。

# 泛型算法

## 参数绑定

有些标准库算法只接受一个包含一个参数的调用对象，但有时候我们想要传递给算法的函数包含两个参数。

如下例，我们希望通过 filter 函数将容器中小于一个给定值的元素设置为 0

### filter 函数

```
void filter(int &a, int n) {  
    a = a < n ? 0 : a;  
}
```

同时我们希望通过 for\_each 算法来遍历元素，但是该算法第三个参数接受只包含一个参数的可调用对象。此时我们可以通过 lambda 表达式来实现：

### filter 函数

```
vector<int> vi = { 3, 7, 1, 11, 3, 3, 2 };  
int n = 3;  
for_each(vi.begin(), vi.end(), [n](int &i) { i = (i  
    < n ? 0 : i);});
```

### 说明

通过 lambda 捕获列表，我们可以从外部设定 n 的值



# 泛型算法

## 参数绑定

有些标准库算法只接受一个包含一个参数的调用对象，但有时候我们想要传递给算法的函数包含两个参数。

如下例，我们希望通过 filter 函数将容器中小于一个给定值的元素设置为 0

### filter 函数

```
void filter(int &a, int n) {  
    a = a < n ? 0 : a;  
}
```

同时我们希望通过 for\_each 算法来遍历元素，但是该算法第三个参数接受只包含一个参数的可调用对象。此时我们可以通过 lambda 表达式来实现：

### filter 函数

```
vector<int> vi = { 3, 7, 1, 11, 3, 3, 2 };  
int n = 3;  
for_each(vi.begin(), vi.end(), [n](int &i) { i = (i  
    < n ? 0 : i);});
```

### 问题

如果坚持使用 filter 呢？

# 泛型算法

## 参数绑定

如果我们依然坚持用 `filter` 函数代替 `lambda` 表达式，我们可以使用标准库 `bind`

### bind 使用格式

```
auto newFun = bind (fun, arg_list);
```

### bind

`bind` 函数接受一个可调用对象，生成一个新的可调用对象来仿造原调用对象的参数列表。

### 说明

- `fun` 是一个已定义的调用对象，`newFun` 是 `fun` 的仿造者
- `arg_list` 是 `fun` 的参数列表
- `arg_list` 可能包含一些名为 `_n` 的参数，他们是 `newFun` 的参数，`n` 的值表示在 `newFun` 参数列表中的位置。
- 当我们调用 `newFun` 时，`newFun` 会调用 `fun`，并把 `arg_list` 中的参数传递给 `fun`。

# 泛型算法

## 参数绑定

下面我们根据 `filter` 函数仿造一个新的调用对象 `uf`:

bind 使用案例 1——根据 `filter` 仿造新对象 `uf`

```
auto uf = bind(filter, std::placeholders::_1, n);
```

bind 使用案例 1——调用 `for_each`

```
for_each(vi.begin(), vi.end(), uf);
```

### 说明

- 仿函数 `uf` 包含一个参数 `_1`
- 调用 `uf` 的时候，它会调用 `filter` 函数来完成实际的工作，并把参数 `_1` 和参数 `n` 传递给 `filter` 函数。
- 调用 `for_each` 算法时，它会将给定的元素传递给 `uf`，`uf` 将这个元素和 `n` 传递给 `filter`，最终完成 `filter` 函数的调用

# 泛型算法

## 参数绑定

默认情况下，bind 函数中不是占位符的参数将以**拷贝**的方式传递给可调用对象。如果需要**传递引用**，可以使用标准库函数 ref：

### bind 使用案例 3——传递引用

```
void sum(int a, int &s){  
    s += a;  
}
```

### 说明

- sum 将第一个函数的值累加到与形参 s 绑定的实参中
- ref 函数返回一个包含 s 的引用的对象

### bind 使用案例 3——传递引用

```
int s = 0; //保存累加和  
for_each(vi.begin(), vi.end(), bind(sum, std::placeholders::_1, ref(s)));
```

### 提示

标准库提供类似 ref 的 cref 函数，其返回包含**const 引用类型**的对象

# 泛型算法

使用 function

前述内容中用于 LargeData 对象比较的调用对象，例如函数、函数对象、bind 函数创建的对象等等虽然使用方式不同但都具有**相同的调用形式**：

## 用于比较的可调用对象的形式

```
bool(LargeData*, LargeData*)
```

## function

通用多态函数封装器，其实例能存储、复制及调用任何**可调用 (Callable)** 目标

## 使用 function1

```
using CallType = bool(LargeData*, LargeData*);  
function<CallType> f1 = Less; //函数  
function<CallType> f2 = Compare(); //函数对象  
function<CallType> f3 = [](const LargeData *a, const  
    LargeData *b) {return a->m_id < b->m_id; }; //  
    lambda  
function<CallType> f4 = //bind函数  
bind(Less, std::placeholders::_2, std::placeholders  
    ::_1);
```

## 说明

- 左上的调用形式是一个函数类型，接受两个 LargeData 指针类型的形参，返回值为 bool
- 左下代码中定义了 bool(LargeData\*,LargeData\*) 调用形式的 CallType 类型

# 泛型算法

使用 `function`

本章结束