

C++_STL 总结

郑华

2019 年 3 月 27 日

目录

第一章 容器-Container	5
1.1 结构	5
1.2 容器底层数据结构实现	6
1.3 重要网站	7
1.4 <code>priority_queue<int, vector<int>, greater<int> ></code>	8
1.4.1 特点	8
1.4.2 用途	8
1.5 Vector	9
1.5.1 关于内存泄露的检验	9
1.5.2 细节	10
1.6 List	12
1.6.1 特点	12
1.6.2 用途	12
1.7 Set	13
1.7.1 排序	13
1.7.2 特点	13
1.7.3 用途	13
1.8 Map	14
1.8.1 特点	14
1.8.2 使用	14
1.9 Hash 序列-> <code>unordered_map</code> or <code>set</code>	16
1.9.1 特点	16
1.9.2 Hash tables	16
1.9.3 实现	17
1.9.4 用法	18
第二章 迭代器-Iterator	19
2.1 性质	19
2.2 使用方法	19
2.3 类别	20
2.4 删除	20

2.5	Iterator 失效概要	21
2.6	Insertion 失效总结	22
2.6.1	Sequence containers	22
2.6.2	Associative containers	22
2.6.3	Unsorted associative containers	23
2.6.4	Container adaptors	23
2.7	Eraser 失效总结	23
2.7.1	Sequence containers	23
2.7.2	Associative containers	23
2.7.3	Unsorted associative containers	23
2.7.4	Container adaptors	24
2.8	Resizing 失效总结	24
2.8.1	Sequence containers	24
2.8.2	Associative containers	24
2.8.3	Unsorted associative containers	24
2.8.4	Container adaptors	24
2.9	Note	25
2.10	插入迭代器	25
2.11	输入输出迭代器	27
2.12	参考	28
第三章	仿函数-Function	29
3.1	std::function	29
3.2	std::bind	30
第四章	算法-Algorithm	35
4.1	查找算法	36
4.2	排序和通用算法	37
4.3	删除和替换算法	38
4.4	排列组合算法	38
4.5	生成和异变算法	39
4.6	关系算法	39
4.7	集合算法	40
4.8	堆算法	40
4.9	算术算法	40
第五章	参考	41

第一章 容器-Container

1.1 结构

STL 主要包括如下组件：

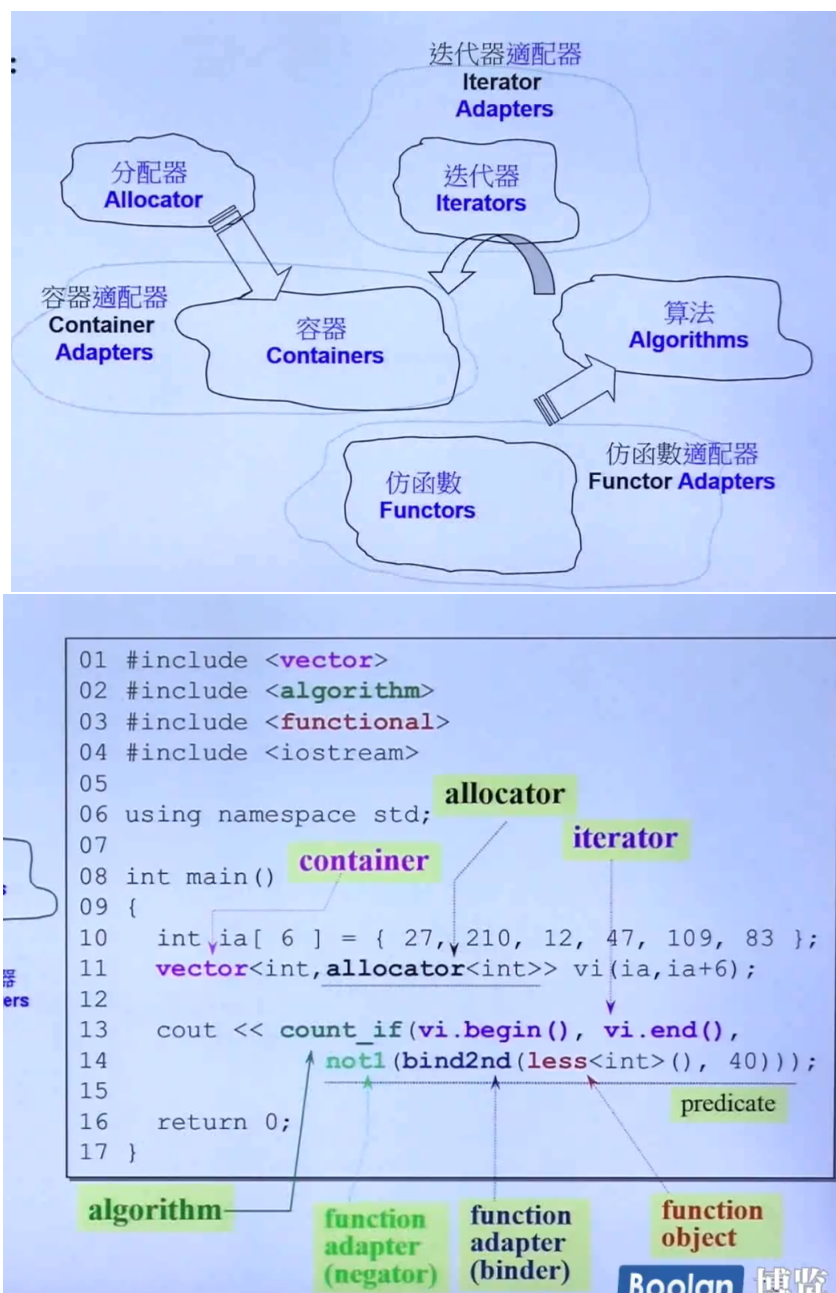


Fig 1.1: STL 组成

1.2 容器底层数据结构实现

0.tuple tuple 元组是一个固定大小的不同类型值的集合，可以用来代替简单的结构体

1.vector 底层数据结构为**数组**，支持快速随机访问

array 专门的数组.. 固定大小

2.list 底层数据结构为**双向链表**，支持快速增删

forward_list 单向链表..

3.deque 底层数据结构为一个**中央控制器**和多个**缓冲区**，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问

deque 是一个双端队列 (double-ended queue)，也是在堆中保存内容的. 它的保存形式如下:

[堆 1] -> [堆 2] -> [堆 3] -> ...

每个堆保存好几个元素, 然后堆和堆之间有指针指向, 看起来像是 list 和 vector 的结合品.

4.stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

5.queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

(stack 和 queue 其实是适配器, 而不叫容器, 因为是对容器的再封装)

6.priority_queue 底层数据结构一般为 **vector** 为底层容器，堆 heap 为处理规则来管理底层容器实现

7.set 底层数据结构为**红黑树**，有序，不重复

8.multiset 底层数据结构为**红黑树**，有序，可重复

9.map 底层数据结构为**红黑树**，有序，不重复

10.multimap 底层数据结构为**红黑树**，有序，可重复

11.unordered_set 底层数据结构为 **hash 表**，无序，不重复

12.unordered_multiset 底层数据结构为 **hash 表**，无序，可重复

13.unordered_map 底层数据结构为 **hash 表**，无序，不重复

14.`unordered_multimap` 底层数据结构为 hash 表，无序，可重复

红黑树 参考文献: <http://blog.csdn.net/chenhua jie123/article/details/11951777>

1.3 重要网站

cpp: cplusplus.com

cppReference: cppreference.com

glibcC++: gcc.gnu.org

1.4 `priority_queue<int, vector<int>, greater<int> >`

1.4.1 特点

- 允许重复元素
- 一种适配器 [包装了底层容器如 `vector` 的高层抽象]
- 可 `pop()`、`push()`
- `pop()` 的元素永远为当前序列里中最小或最大的。

1.4.2 用途

1.5 Vector

1.5.1 关于内存泄露的检验

随着用 string 越来越多，有的时候你会发现 string 的内存管理的问题，存在内存暂时泄露的问题。这个内存泄露与我们常规说的内存泄露问题不一样。它不是真的内存泄露，在程序结束的时候，内存还是会释放掉的，但是在程序的运行过程中，内存被 string or vector 对象占用着。比如你有个 string 对象保存了一个比较大的网页，用完了之后，你想通过 clear 来回收这个对象的内存，你这样是做不到的。因为 string 的 clear 并不会真正回收 string 对象分配内存。那要怎么做才能做到呢？

vector 与 string 类似，而 string 比较少。

```
vector<int> test;

cout << test.capacity() << endl;
cout << "after_pushOneData_test_capacity_is_" << test.capacity() << endl;

for (int i = 0; i < 100; ++i)
test.push_back(i);

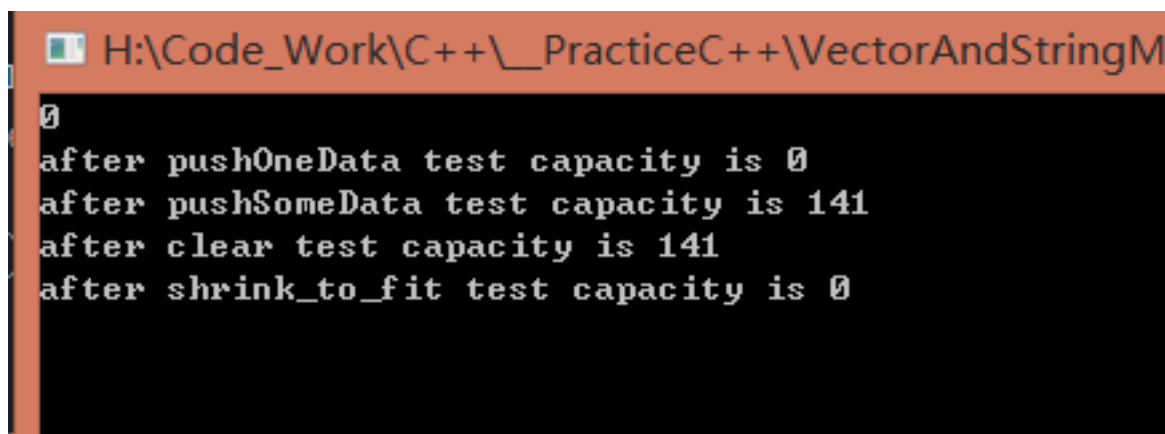
cout << "after_pushSomeData_test_capacity_is_" << test.capacity() << endl;

test.clear();

cout << "after_clear_test_capacity_is_" << test.capacity() << endl;

test.shrink_to_fit();
//test.swap(vector<int>()); 同样可以解决内存泄露问题.. 交换一个空的临时对象.. 这样就可以
    利用对象在离开作用域时析构的特性了

cout << "after_shrink_to_fit_test_capacity_is_" << test.capacity() << endl;
```

A screenshot of a Windows command prompt window with a dark background. The title bar shows the file path "H:\Code_Work\C++_PracticeC++\VectorAndStringM". The output of the program is displayed in white text, showing the capacity of a vector at different stages: 0, 141, 141, and 0.

```
H:\Code_Work\C++\_PracticeC++\VectorAndStringM
0
after pushOneData test capacity is 0
after pushSomeData test capacity is 141
after clear test capacity is 141
after shrink_to_fit test capacity is 0
```

Fig 1.2: vector 内存泄露检测

Notice vector 依旧 string clear 后，需要调用.shrink_to_fit 才可将内存清除，防止内存泄露，或者与一个同类型空容器交换以释放内存。

1.5.2 细节

push_back 与 emplace_back

<http://blog.csdn.net/yockie/article/details/52674366>

```
#include <vector>
#include <string>
#include "time_interval.h"

class Foo {
public:
    Foo(std::string str) : name(str) {
        std::cout << "constructor" << std::endl;
    }
    Foo(const Foo& f) : name(f.name) {
        std::cout << "copy_constructor" << std::endl;
    }
    Foo(Foo&& f) : name(std::move(f.name)){
        std::cout << "move_constructor" << std::endl;
    }

private:
    std::string name;
};

int main()
{
    std::vector<Foo> v;
    int count = 10000000;
    v.reserve(count); //预分配十万大小，排除掉分配内存的时间

    {
        TIME_INTERVAL_SCOPE("push_back_T:");
        Foo temp("ceshi");
        v.push_back(temp); // push_back(const T&), 参数是左值引用
        //打印结果:
        //constructor
        //copy constructor
    }

    v.clear();
    {
        TIME_INTERVAL_SCOPE("push_back_move(T):");
        Foo temp("ceshi");
        v.push_back(std::move(temp)); // push_back(T &&), 参数是右值引用
    }
}
```

```

        //打印结果:
        //constructor
        //move constructor
    }

v.clear();
{
    TIME_INTERVAL_SCOPE("push_back(T&&):");
    v.push_back(Foo("ceshi")); // push_back(T &&), 参数是右值引用
    //打印结果:
    //constructor
    //move constructor
}

v.clear();
{
    std::string temp = "ceshi";
    TIME_INTERVAL_SCOPE("push_back(string):");
    v.push_back(temp); // push_back(T &&), 参数是右值引用
    //打印结果:
    //constructor
    //move constructor
}

v.clear();
{
    std::string temp = "ceshi";
    TIME_INTERVAL_SCOPE("emplace_back(string):");
    v.emplace_back(temp); // 只有一次构造函数, 不调用拷贝构造函数, 速度最快
    //打印结果:
    //constructor
}
}

```

1.6 List

1.6.1 特点

- 允许重复元素
- 实现了基本的 `==` 和 `!=` 等 (如果两个链表里的元素都相等返回 `true`)
- 底层数据结构为双向链表
- 可前 (`push_front()`)、后 (`push_back()`)、任意位置插入 (`insert()`)
- 当然就有 `pop` 了

1.6.2 用途

1.7 Set

1.7.1 排序

<http://blog.csdn.net/wangran51/article/details/8836160>

1.7.2 特点

- 无重复元素
- 元素默认升序排序【1,2,3】
- 高效的查找：红黑二叉检索树
- 不可以修改元素
- 快捷的删除

1.7.3 用途

主要用于检索数据

1.8 Map

1.8.1 特点

- 无重复元素
- 高效的查找：红黑二叉检索树，查找和添加的复杂度都为 $O(\log(n))$, `unordered_map` 使用 hash 表作为基本的存储结构, $O(1)$ 。
- 可以修改元素
- 快捷的删除

1.8.2 使用

map 最基本的构造函数

- `map<string , int >mapstring;`
- `map< char ,string>mapchar;`
- `map<int ,char >mapint;`

map 添加数据

- `map<int ,string> maplive;`
1. `maplive.insert(pair<int,string>(102,"active"));`
 2. `maplive.insert(map<int,string>::value_type(321,"hai"));`
 3. `maplive[112]="April";` map 中最简单最常用的插入添加！

map 中元素的查找

- 定义迭代器准备保存查找返回位置:`map<int ,string >::iterator l_it;`
- Find 函数: `l_it=maplive.find(112);`
- 结果判断: `if(l_it==maplive.end()) not find`

map 中元素的删除

先查找，返回迭代器位置，`erase`。

map 的 sort 问题

Map 中的元素是自动按 key 升序排序，所以不能对 map 用 `sort` 函数

其他操作函数

- `begin()` 返回指向 `map` 头部的迭代器
- `clear()` 删除所有元素
- `count()` 返回指定元素出现的次数
- `rbegin()` 返回一个指向 `map` 尾部的逆向迭代器
- `size()` 返回 `map` 中元素的个数

1.9 Hash 序列-> unordered_map or set

1.9.1 特点

- 无重复元素
- 高效的查找：查找和添加复杂度均为 $O(1)$
- 可以修改元素
- 快捷的删除

1.9.2 Hash tables

Including hash tables (unordered associative containers) in the C++ standard library is one of the most recurring requests. It was not adopted in C++03 due to time constraints only. Although hash tables are less efficient than a balanced tree in the worst case (in the presence of many collisions), they perform better in many real applications.

Collisions are managed only via linear chaining because the committee didn't consider it to be opportune to standardize solutions of open addressing that introduce quite a lot of intrinsic problems (above all when erasure of elements is admitted). To avoid name clashes with non-standard libraries that developed their own hash table implementations, the prefix “**unordered**” was used *instead of* “**hash**”.

The new library has four types of hash tables, differentiated by whether or not they accept elements with the same key (**unique keys or equivalent keys**), and whether they map each key to an associated value. They correspond to the four existing binary-search-tree-based associative containers, with an `unordered_` prefix.

Type of hash table	Associated values	Equivalent keys
<code>std::unordered_set</code>	No	No
<code>std::unordered_multiset</code>	No	Yes
<code>std::unordered_map</code>	Yes	No
<code>std::unordered_multimap</code>	Yes	Yes

Fig 1.3: HashTable - 4types

```
#include <iostream>
#include <string>
#include <unordered_map>

int main()
```



```

{
    std::unordered_map<std::string, int> months;
    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    months["may"] = 31;
    months["june"] = 30;
    months["july"] = 31;
    months["august"] = 31;
    months["september"] = 30;
    months["october"] = 31;
    months["november"] = 30;
    months["december"] = 31;
    std::cout << "september_>" << months["september"] << std::endl;
    std::cout << "april_>" << months["april"] << std::endl;
    std::cout << "december_>" << months["december"] << std::endl;
    std::cout << "february_>" << months["february"] << std::endl;
    return 0;
}

```

1.9.3 实现

HashMap 基于 hash table（哈希表）。哈希表最大的优点，就是把数据的存储和查找消耗的时间大大降低，几乎可以看成是常数时间；而代价仅仅是消耗比较多的内存。然而在当前可利用内存越来越多的情况下，用空间换时间的做法是值得的

其基本原理是：使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数，也叫做散列函数），使得每个元素的关键字都与一个函数值（即数组下标，hash 值）相对应，于是用这个数组单元来存储这个元素；也可以简单的理解为，按照关键字为每一个元素“分类”，然后将这个元素存储在相应“类”所对应的地方，称为桶。

但是，不能够保证每个元素的关键字与函数值是一一对应的，因此极有可能出现对于不同的元素，却计算出了相同的函数值，这样就产生了“冲突”，换句话说，就是把不同的元素分在了相同的“类”之中。总的来说，“直接定址”与“解决冲突”是哈希表的两大特点

HashMap，首先分配一大片内存，形成许多桶。是利用 hash 函数，对 key 进行映射到不同区域（桶）进行保存。

插入过程是：

1. 得到 key
2. 通过 hash 函数得到 hash 值
3. 得到桶号（一般都为 hash 值对桶数求模）
4. 存放 key 和 value 在桶内

取值过程是：

1. 得到 key
2. 通过 hash 函数得到 hash 值
3. 得到桶号 (一般都为 hash 值对桶数求模)
4. 比较桶的内部元素是否与 key 相等，若都不相等，则没有找到。
5. 取出相等的记录的 value。

1.9.4 用法

与 map 大体一致。

第二章 迭代器-Iterator

2.1 性质

- 它是一个访问容器对象的技术手段，通过迭代器可以访问容器中的对象
- 功能类似于指针 或者 数组下标，通过指针加减与数组下标运算获得下一数据成员
- 在实现上，迭代器可以是指针，但并不必须是指针，也不必总是使用数据对象的地址。只要能够访问到数据对象就可以了。

2.2 使用方法

- 声明迭代器变量
- 使用引领操作符访问迭代器指向的当前目标对象
- 使用递增操作符获得下一对象的访问权
- 若迭代器新值超出容器的元素范围，类似指针值变成 NULL，目标对象不可引用

```
// 声明如下:
vector<T>::iterator it;
list<T>::iterator it;
deque<T>::iterator it;

#include <iostream>
#include <algorithm>
using namespace std;
const int size = 16;
int main()
{
    int a[size];
    for( int i = 0; i < size; ++i ) a[i] = i;
    int key = 7;
    int * ip = find( a, a + size, key );
    if( ip == a + size ) // 不要使用NULL做指针测试，直接使用过尾值 []
        cout << key << " not found." << endl;
    else
```

```

        cout << key << " found." << endl;
    return 0;
}

```

2.3 类别

表 2.1: 迭代器类别

类别	功能与使用
输入	从容器中读取元素
输出	向容器中写入元素
正向	组合输入迭代器和输出迭代器的功能，并保留在容器中的位置
双向	组合正向迭代器和逆向迭代器的功能，支持多遍算法
随机	组合双向迭代器的功能与直接访问容器中任何元素的功能，即可向前向后跳过任意个元素

表 2.2: 不同迭代器对应的可执行操作

类别	可使用的操作
所有	<code>++p, p++</code> [位置自增运算]
输入	<code>*p</code> [作为右值], <code>p = q</code> [将一个迭代器赋给另一个迭代器], <code>p == q</code> [相等比较运算，但不能比较大小]
输出	<code>*p</code> [作为左值], <code>p = q</code> [将一个迭代器赋给另一个迭代器]
正向	提供输入输出迭代器的所有功能
双向	<code>-p, p-</code> [位置自减运算]
随机	<code>p+=i</code> [将迭代器递增 <code>i</code> 位], <code>p+i</code> [在 <code>p</code> 位加 <code>i</code> 位后的迭代器], <code>p[i]</code> [返回 <code>p</code> 位元素偏离 <code>i</code> 位的元素引用], <code>p < p1</code> [如果迭代器 <code>p</code> 的位置在 <code>p1</code> 前, 返回 <code>true</code> , 否则返回 <code>false</code>]

2.4 删除

vector 用到 `erase` 的话, 那必须用迭代器了, 那么以下就要注意了

- 在执行 `erase` 后, 不可以进行 `++pos` 操作, 否则会报错
- 在不执行 `erase` 时, 要进行 `++pos` 进行前进

```

for(auto i = 1; i < nums.size(); )
{
    flag = true;
    if(nums[i-1] == nums[i])

```

表 2.3: 各容器对应的迭代器类别

容器类别	可使用的迭代器
vector	随机
deque	随机
list	双向
set	双向 multi 同
map	双向 multi 同
stack	不支持
queue	不支持
priority_queue	不支持

```

{
    ++size;
    if(size > 2)
    {
        --lenth;
        nums.erase(pos); //这块执行pos 的话就不用再移位置了
        flag = false;
    }
}
else
{
    size = 1;
}
if(flag) //否则，移位置前进
{
    ++i;
    ++pos;
}
}

```

2.5 Iterator 失效概要

Golden Rule 是：尽量不要使用容器的插入删除操作之前的迭代器。

对于 vector ,deque, list, 一种可行的方式是：

```

std::vector<int>::iterator it = my_container.begin();
for (it != my_container.end(); /**blank*/ ) {
    if (*it % 2 == 1) {
        it = my_container.erase(it);
    }
    else{
        it++;
    }
}

```

```

    }
}

// Method Two --
std::vector<int>::iterator it = my_container.begin();
for (it != my_container.end(); /**blank*/ ) {
    if (*it % 2 == 1) {
        my_container.erase(it++);
    }
    else{
        it++;
    }
}
}
/*
my_container.erase(it++) 巧妙得在执行erase()之前，it 先自增，指向被删除元素后面的元
素，而给erase()传递的是未自增的it迭代器，以定位要删除的元素。
如果元素的值为奇数，则删除此元素，it指向下一个元素，如果元素的值为偶数，则检查下一个
元素的值。整个迭代过程中迭代器就不会失效了。
*/

```

2.6 Insertion 失效总结

2.6.1 Sequence containers

- **vector**: all iterators and references before the point of insertion are unaffected, unless the new container size is greater than the previous capacity (in which case all iterators and references are invalidated)
- **deque**: all iterators and references are invalidated, unless the inserted member is at an end (front or back) of the deque (in which case all iterators are invalidated, but references to elements are unaffected)
- **list**: all iterators and references unaffected
- **forward_list**: all iterators and references unaffected (applies to `insert_after`)
- **array**: (n/a)

2.6.2 Associative containers

- **[multi]{set,map}**: all iterators and references unaffected

2.6.3 Unsorted associative containers

- `unordered_[multi]{set,map}`: all iterators invalidated when rehashing occurs, but references unaffected [23.2.5/8]. Rehashing does not occur if the insertion does not cause the container's size to exceed $z * B$ where z is the maximum load factor and B the current number of buckets.

2.6.4 Container adaptors

- `stack`: inherited from underlying container
- `queue`: inherited from underlying container
- `priority_queue`: inherited from underlying container

2.7 Eraser 失效总结

2.7.1 Sequence containers

- `vector`: every iterator and reference at or after the point of erase is invalidated
- `deque`: erasing the last element invalidates only iterators and references to the erased elements and the past-the-end iterator; erasing the first element invalidates only iterators and references to the erased elements; erasing any other elements invalidates all iterators and references (including the past-the-end iterator)
- `list`: only the iterators and references to the erased element is invalidated
- `forward_list`: only the iterators and references to the erased element is invalidated (applies to `erase_after`)
- `array`: (n/a)

2.7.2 Associative containers

- `[multi]{set,map}`: only iterators and references to the erased elements are invalidated

2.7.3 Unsorted associative containers

- `unordered_[multi]{set,map}`: all iterators invalidated when rehashing occurs, but references unaffected [23.2.5/8]. Rehashing does not occur if the insertion does not cause the container's size to exceed $z * B$ where z is the maximum load factor and B the current number of buckets.

2.7.4 Container adaptors

- `stack`: inherited from underlying container
- `queue`: inherited from underlying container
- `priority_queue`: inherited from underlying container

2.8 Resizing 失效总结

2.8.1 Sequence containers

- `vector`: as per insert/erase
- `deque`: as per insert/erase
- `list`: as per insert/erase
- `forward_list`: as per insert/erase
- `array`: (n/a)

2.8.2 Associative containers

- `[multi]{set,map}`: only iterators and references to the erased elements are invalidated

2.8.3 Unsorted associative containers

- `unordered_[multi]{set,map}`: all iterators invalidated when rehashing occurs, but references unaffected [23.2.5/8]. Rehashing does not occur if the insertion does not cause the container's size to exceed $z * B$ where z is the maximum load factor and B the current number of buckets.

2.8.4 Container adaptors

- `stack`: inherited from underlying container
- `queue`: inherited from underlying container
- `priority_queue`: inherited from underlying container

2.9 Note

- **Unless otherwise specified** (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to **a library function shall not invalidate iterators** to, or change the values of, objects within that container.
- **no swap()** function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [Note: **The end() iterator** does not refer to any element, so it **may be invalidated**. —end note]
- Other than the above caveat regarding swap(), it's not clear whether "end" iterators are subject to the above listed per-container rules; you should assume, anyway, that they are
- **vector** and all unordered associative containers support **reserve(n)** which guarantees that no automatic resizing will occur at least until the size of the container grows to **n**. Caution should be taken with unordered associative containers because a future proposal will allow the specification of a minimum load factor, which would allow rehashing to occur on **insert** after enough **erase** operations reduce the container size below the minimum; the guarantee should be considered potentially void after an **erase**.

2.10 插入迭代器

迭代器是一个纯粹抽象概念：任何东西，只要其行为类似迭代器，它就是一个迭代器。C++ 标准库提供了数个预先定义的特殊迭代器，即迭代器适配器（**iterator adapters**）。它们不仅起辅助作用，还能赋予整个迭代器抽象概念更强大的能力。

说明：适配器是使一事物的行为类似于另一事物的行为的一种机制。

插入器是一种迭代器适配器，带有一个容器参数，并**生成**一个迭代器，用于在指定容器中插入元素。通过插入迭代器赋值时，迭代器将会插入一个新元素。C++ 提供了三种插入器，其差别在于插入元素位置不同。

1. **back_inserter**: 创建使用**push_back** 实现插入的迭代器

back_inserter 内部调用**push_back**，在容器末尾插入元素。因此，只有在提供有**push_back** 成员函数的容器中才能使用。这样的容器有：**vector, deque, list**。元素排列次序和安插次序相同

2. **front_inserter**: 创建使用**push_front** 实现插入的迭代器

front_inserter 内部调用**push_front**，将元素安插于容器最前端。因此，只有在提供有**push_front** 成员函数的容器中才能使用。这样的容器有：**deque, list**。元素排列次序和安插次序相反

3. **insert**: 创建使用**insert** 实现插入的迭代器

Insertter 内部调用insert，在它的迭代器实参所标明的位置前面插入元素。所有STL 容器都提供insert 成员函数，因此这是唯一可用于关联容器上的插入迭代器。元素排列次序和安插次序相同

```
#include <iostream>
#include <vector>
#include <list>
#include <iterator>

using namespace std;

template<typename T>
void PrintElements(T c)
{
    T::const_iterator itr = c.begin();
    while(itr != c.end())
    {
        cout<<*itr++<<" ";
    }
}

int main()
{
    vector<int> vecSrc;
    list<int> vecDest;

    for(vector<int>::size_type i=0; i<3; ++i)
    {
        vecSrc.push_back(i);
    }

    /*
        list<int> coll1;
        vector<int> coll2;
        copy(coll1.begin(), coll1.end(), back_inserter(coll2));
    */
    copy(vecSrc.begin(),vecSrc.end(),back_insert_iterator<list<int> >(vecDest));
    PrintElements(vecDest);
    cout<<endl;

    copy(vecSrc.begin(),vecSrc.end(),front_insert_iterator<list<int> >(vecDest));
    PrintElements(vecDest);
    cout<<endl;

    copy(vecSrc.begin(),vecSrc.end(),insert_iterator<list<int> >(vecDest,++vecDest.
        begin())));
    PrintElements(vecDest);
```

```

        return 0;
    }

    // 2 0 1 - 2 1 0 - 0 1 2

```

2.11 输入输出迭代器

标准程序库定义有供输入及输出用的 `istream_iterator` 类，称为 `istream_iterator` 和 `ostream_iterator`，分别支持单一型别的元素读取和写入

1. `ostream_iterator`:

```

// ostream_iterator example
#include <iostream> // std::cout
#include <iterator> // std::ostream_iterator
#include <vector> // std::vector
#include <algorithm> // std::copy

int main () {
    std::vector<int> myvector;
    for (int i=1; i<10; ++i) myvector.push_back(i*10);

    std::ostream_iterator<int> out_it (std::cout, "\n");
    std::copy ( myvector.begin(), myvector.end(), out_it );
    std::unique_copy(myvector.begin(), myvector.end(), out_it);
    return 0;
}

// 10, 20, 30, 40, 50, 60, 70, 80, 90,

```

2. `istream_iterator`:

```

// istream_iterator example
#include <iostream> // std::cin, std::cout
#include <iterator> // std::istream_iterator

int main () {
    double value1, value2;
    std::cout << "Please, insert two values: ";

    std::istream_iterator<double> eos; // end-of-stream iterator
    std::istream_iterator<double> iit (std::cin); // stdin iterator

    if (iit!=eos) value1=*iit;

```

```
++iit;
if (iit!=eos) value2=*iit;

std::cout << value1 << "*" << value2 << "=" << (value1*value2) << '\n';

return 0;
}

// Please, insert two values: 2 32 2*32=64
```

2.12 参考

<http://www.cnblogs.com/dongzhiquan/archive/2011/01/05/1994523.html>

<http://www.cnblogs.com/blueoverflow/p/4923523.html>

失效重要:<http://stackoverflow.com/questions/6438086/iterator-invalidation-rules>

<http://lib.csdn.net/article/cplusplus/28411>

C++ STL 插入器: <http://www.linuxidc.com/Linux/2015-02/113456.htm>

<http://blog.csdn.net/bonchoix/article/details/8062483>

第三章 仿函数-Function

3.1 std::function

在 C++ 中，可调用实体主要包括函数，函数指针，函数引用，可以隐式转换为函数指定的对象，或者实现了 `operator()` 的对象。C++0x 中，新增加了一个 `std::function` 对象，`std::function` 对象是对 C++ 中现有的可调用实体的一种类型安全的包裹（我们知道像函数指针这类可调用实体，是类型不安全的）。`std::function` object 最大的用处就是在实现函数回调，使用者需要注意，它不能被用来检查相等或者不相等

```
#include<iostream> // std::cout
#include<functional> // std::function

void func(void)
{
    std::cout << __FUNCTION__ << std::endl;
}

class Foo
{
public:
    static int foo_func(int a)
    {
        std::cout << __FUNCTION__ << "(" << a << ")_->:_";
        return a;
    }
};

class Bar
{
public:
    int operator() (int a)
    {
        std::cout << __FUNCTION__ << "(" << a << ")_->:_";
        return a;
    }
};

int main()
```

```

{
    // 绑定普通函数
    std::function<void(void)> fr1 = func;
    fr1();

    // 绑定类的静态成员函数
    std::function<int(int)> fr2 = Foo::foo_func;
    std::cout << fr2(100) << std::endl;

    // 绑定仿函数
    Bar bar;
    fr2 = bar;
    std::cout << fr2(200) << std::endl;

    return 0;
}

```

3.2 std::bind

bind 是这样一种机制，它可以预先把指定可调用实体的某些参数绑定到已有的变量，产生一个新的可调用实体，这种机制在回调函数的使用过程中也颇为有用。

C++11 中提供了 std::bind，可以说是一种飞跃的提升，bind 本身是一种延迟计算的思想，它本身可以绑定普通函数、全局函数、静态函数、类静态函数甚至是类成员函数。

std::bind 用来将可调用对象与其参数一起进行绑定。绑定后可以使用 **std::function** 进行保存，并延迟到我们需要的时候调用：

- bind 预先绑定的参数需要传具体的变量或值进去，对于预先绑定的参数，是 pass-by-value 的
- 对于不事先绑定的参数，需要传 std::placeholders 进去，从 _1 开始，依次递增。placeholder 是 pass-by-reference 的
- bind 的返回值是可调用实体，可以直接赋给 std::function 对象
- 对于绑定的指针、引用类型的参数，使用者需要保证在可调用实体调用之前，这些参数是可用的

在绑定部分参数的时候，通过使用 **std::placeholders** 来决定空位参数将会是调用发生时的第几个参数。

```

#include <iostream>
#include <functional>
using namespace std;

int TestFunc(int a, char c, float f)

```

```

{
    cout << a << endl;
    cout << c << endl;
    cout << f << endl;

    return a;
}

int main()
{
    auto bindFunc1 = bind(TestFunc, std::placeholders::_1, 'A', 100.1);
    bindFunc1(10);

    cout << "=====\n";

    auto bindFunc2 = bind(TestFunc, std::placeholders::_2, std::placeholders::_1,
        100.1);
    bindFunc2('B', 10);

    cout << "=====\n";

    auto bindFunc3 = bind(TestFunc, std::placeholders::_2, std::placeholders::_3, std
        ::placeholders::_1);
    bindFunc3(100.1, 30, 'C');

    return 0;
}

\\ Example 2:
int add1(int i, int j, int k) {
    return i + j + k;
}

class Utils {
public:
    Utils(const char* name) {
        strcpy(_name, name);
    }

    void sayHello(const char* name) const {
        std::cout << _name << "say:hello" << name << std::endl;
    }

    static int getId() {
        return 10001;
    }
}

```

```

int operator()(int i, int j, int k) const {
    return i + j + k;
}

private:
char _name[32];
};

int main(void) {

    // 绑定全局函数
    auto add2 = std::bind(add1, std::placeholders::_1, std::placeholders::_2, 10);
    // 函数add2 = 绑定add1函数, 参数1不变, 参数2不变, 参数3固定为10.
    std::cout << typeid(add2).name() << std::endl;
    std::cout << "add2(1,2)_=_ " << add2(1, 2) << std::endl;

    std::cout << "\n-----" << std::endl;

    // 绑定成员函数
    Utils utils("Vicky");
    auto sayHello = std::bind(&Utils::sayHello, utils/*调用者*/, std::placeholders::_1
        /*参数1*/);
    sayHello("Jack");

    auto sayHelloToLucy = std::bind(&Utils::sayHello, utils/*调用者*/, "Lucy"/*固定参数
        1*/);
    sayHelloToLucy();

    // 绑定静态成员函数
    auto getId = std::bind(&Utils::getId);
    std::cout << getId() << std::endl;

    std::cout << "\n-----" << std::endl;

    // 绑定operator函数
    auto add100 = std::bind(&Utils::operator (), utils, std::placeholders::_1, std::
        placeholders::_2, 100);
    std::cout << "add100(1,_2)_=_ " << add100(1, 2) << std::endl;

    // 注意: 无法使用std::bind()绑定一个重载函数

    return 0;
}

```

从上面的代码可以看到, bind 能够在绑定时候就同时绑定一部分参数, 未提供的参数则使用占位符表示, 然后在运行时传入实际的参数值。

PS: 绑定的参数将会以值传递的方式传递给具体函数, 占位符将会以引用传递。

bind 实现 <https://www.cnblogs.com/lit10050528/p/7017768.html>

参考 : <http://blog.csdn.net/eclipser1987/article/details/24406203>

第四章 算法-Algorithm

算法 (Algorithm)，是用来操作容器中的数据的模板函数。例如，STL 用 `sort()` 来对一个 `vector` 中的数据进行排序，用 `find()` 来搜索一个 `list` 中的对象，函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用；

STL 算法部分主要由头文件 `<algorithm>`, `<numeric>`, `<functional>` 组成。要使用 STL 中的算法函数必须包含头文件 `<algorithm>`，对于数值算法须包含 `<numeric>`。

4.1 查找算法

表 4.1: 查找算法

函数名	功能与使用
adjacent_find	在 iterator 对标识元素范围内，查找一对相邻重复元素，找到则返回指向这对元素的第一个元素的 ForwardIterator。否则返回 last。
binary_search	在有序序列中查找 value，找到返回 true。重载的版本实用指定的比较函数对象或函数指针来判断相等
count	利用等于操作符，把标志范围内的元素与输入值比较，返回相等元素个数
count_if	利用输入的操作符，对标志范围内的元素进行操作，返回结果为 true 的个数
equal_range	功能类似 equal，返回一对 iterator，第一个表示 lower_bound，第二个表示 upper_bound
find	利用底层元素的等于操作符，对指定范围内的元素与输入值进行比较。当匹配时，结束搜索，返回该元素的一个 InputIterator
find_end	在指定范围内查找”由输入的另外一对 iterator 标志的第二个序列”的最后一次出现。找到则返回最后一对的第一个 ForwardIterator，否则返回输入的”另外一对”的第一个 ForwardIterator。重载版本使用用户输入的操作符代替等于操作
find_first_of	在指定范围内查找”由输入的另外一对 iterator 标志的第二个序列”中任意一个元素的第一次出现。重载版本中使用了用户自定义操作符
find_if	使用输入的函数代替等于操作符执行 find
lower_bound	返回一个 ForwardIterator，指向在有序序列范围内的可以插入指定值而不破坏容器顺序的第一个位置。重载函数使用自定义比较操作
upper_bound	返回一个 ForwardIterator，指向在有序序列范围内插入 value 而不破坏容器顺序的最后一个位置，该位置标志一个大于 value 的值。重载函数使用自定义比较操作
search	给出两个范围，返回一个 ForwardIterator，查找成功指向第一个范围内第一次出现子序列 (第二个范围) 的位置，查找失败指向 last1。重载版本使用自定义的比较操作
search_n	在指定范围内查找 val 出现 n 次的子序列。重载版本使用自定义的比较操作

4.2 排序和通用算法

表 4.2: 排序和通用算法

函数名	功能与使用
inplace_merge	合并两个有序序列，结果序列覆盖两端范围。重载版本使用输入的操作进行排序
merge	合并两个有序序列，存放到另一个序列。重载版本使用自定义的比较
nth_element	将范围内的序列重新排序，使所有小于第 <i>n</i> 个元素的元素都出现在它前面，而大于它的都出现在后面。重载版本使用自定义的比较操作
partial_sort	对序列做部分排序，被排序元素个数正好可以被放到范围内。重载版本使用自定义的比较操作
partial_sort_copy	与 partial_sort 类似，不过将经过排序的序列复制到另一个容器
partition	对指定范围内元素重新排序，使用输入的函数，把结果为 true 的元素放在结果为 false 的元素之前
random_shuffle	对指定范围内的元素随机调整次序。重载版本输入一个随机数产生操作
reverse	将指定范围内元素重新反序排序
reverse_copy	与 reverse 类似，不过将结果写入另一个容器
rotate	将指定范围内元素移到容器末尾，由 middle 指向的元素成为容器第一个元素
rotate_copy	与 rotate 类似，不过将结果写入另一个容器
sort	以升序重新排列指定范围内的元素。重载版本使用自定义的比较操作
stable_sort	与 sort 类似，不过保留相等元素之间的顺序关系
stable_partition	与 partition 类似，不过不保证保留容器中的相对顺序

4.3 删除和替换算法

表 4.3: 删除和替换算法

函数名	功能与使用
<code>copy</code>	复制序列
<code>copy_backward</code>	与 <code>copy</code> 相同，不过元素是以相反顺序被拷贝
<code>iter_swap</code>	交换两个 Forward Iterator 的值
<code>remove</code>	删除指定范围内所有等于指定元素的元素。注意，该函数不是真正删除函数。内置函数不适合使用 <code>remove</code> 和 <code>remove_if</code> 函数
<code>remove_copy</code>	将所有不匹配元素复制到一个制定容器，返回 OutputIterator 指向被拷贝的末元素的下一个位置
<code>remove_if</code>	删除指定范围内输入操作结果为 <code>true</code> 的所有元素
<code>remove_copy_if</code>	将所有不匹配元素拷贝到一个指定容器
<code>replace</code>	将指定范围内所有等于 <code>vold</code> 的元素都用 <code>vnew</code> 代替
<code>replace_copy</code>	与 <code>replace</code> 类似，不过将结果写入另一个容器
<code>replace_if</code>	将指定范围内所有操作结果为 <code>true</code> 的元素用新值代替
<code>replace_copy_if</code>	与 <code>replace_if</code> ，不过将结果写入另一个容器
<code>swap</code>	交换存储在两个对象中的值
<code>swap_range</code>	将指定范围内的元素与另一个序列元素值进行交换
<code>unique</code>	清除序列中重复元素，和 <code>remove</code> 类似，它也不能真正删除元素。重载版本使用自定义比较操作
<code>unique_copy</code>	与 <code>unique</code> 类似，不过把结果输出到另一个容器

实际上，`remove_if()` 等并没有删除容器中的任何元素，它没有改变 `container.end()`，调用 `remove_if()` 后容器元素个数不会改变!! 删除元素的工作交给 `erase()` 才能真正删除。

4.4 排列组合算法

表 4.4: 排列组合算法

函数名	功能与使用
<code>next_permutation</code>	复制序列
<code>prev_permutation</code>	取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回 <code>false</code> 。重载版本使用自定义的比较操作

4.5 生成和异变算法

表 4.5: 生成和异变算法

函数名	功能与使用
fill	将输入值赋给标志范围内的所有元素
fill_n	将输入值赋给 first 到 first+n 范围内的所有元素
for_each	用指定函数依次对指定范围内所有元素进行迭代访问，返回所指定的函数类型。该函数不得修改序列中的元素
generate	连续调用输入的函数来填充指定的范围
generate_n	与 generate 函数类似，填充从指定 iterator 开始的 n 个元素
transform	将输入的操作作用与指定范围内的每个元素，并产生一个新的序列。重载版本将操作作用在一对元素上，另外一个元素来自输入的另外一个序列。结果输出到指定容器

4.6 关系算法

表 4.6: 关系算法

函数名	功能与使用
equal	如果两个序列在标志范围内元素都相等，返回 true。重载版本使用输入的操作符代替默认的等于操作符
includes	判断第一个指定范围内的所有元素是否都被第二个范围包含，使用底层元素的 < 操作符，成功返回 true。重载版本使用用户输入的函数
lexicographical_compare	比较两个序列。重载版本使用用户自定义比较操作
max	返回两个元素中较大一个。重载版本使用自定义比较操作
max_element	返回一个 Forward Iterator，指出序列中最大的元素。重载版本使用自定义比较操作
min	返回两个元素中较小一个。重载版本使用自定义比较操作
min_element	返回一个 Forward Iterator，指出序列中最小的元素。重载版本使用自定义比较操作
mismatch	并行比较两个序列，指出第一个不匹配的位置，返回一对 iterator，标志第一个不匹配元素位置。如果都匹配，返回每个容器的 last。重载版本使用自定义的比较操作

4.7 集合算法

表 4.7: 集合算法

函数名	功能与使用
<code>set_union</code>	构造一个有序序列，包含两个序列中所有的不重复元素。重载版本使用自定义的比较操作
<code>set_intersection</code>	构造一个有序序列，其中元素在两个序列中都存在。重载版本使用自定义的比较操作
<code>set_difference</code>	构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存在的元素。重载版本使用自定义的比较操作
<code>set_symmetric_difference</code>	构造一个有序序列，该序列取两个序列的对称差集 (并集-交集)

4.8 堆算法

表 4.8: 堆算法

函数名	功能与使用
<code>make_heap</code>	把指定范围内的元素生成一个堆。重载版本使用自定义比较操作
<code>pop_heap</code>	并不真正把最大元素从堆中弹出，而是重新排序堆。它把 <code>first</code> 和 <code>last-1</code> 交换，然后重新生成一个堆。可使用容器的 <code>back</code> 来访问被”弹出”的元素或者使用 <code>pop_back</code> 进行真正的删除。重载版本使用自定义的比较操作
<code>push_heap</code>	假设 <code>first</code> 到 <code>last-1</code> 是一个有效堆，要被加入到堆的元素存放在位置 <code>last-1</code> ，重新生成堆。在指向该函数前，必须先把元素插入容器后。重载版本使用指定的比较操作
<code>sort_heap</code>	对指定范围内的序列重新排序，它假设该序列是个有序堆。重载版本使用自定义比较操作

4.9 算术算法

表 4.9: 算术算法

函数名	功能与使用
<code>accumulate</code>	<code>iterator</code> 对标识的序列段元素之和，加到一个由 <code>val</code> 指定的初始值上。重载版本不再做加法，而是传进来的二元操作符被应用到元素上
<code>partial_sum</code>	创建一个新序列，其中每个元素值代表指定范围内该位置前所有元素之和。重载版本使用自定义操作代替加法
<code>inner_product</code>	对两个序列做内积 (对应元素相乘，再求和) 并将内积加到一个输入的初始值上。重载版本使用用户定义的操作
<code>adjacent_difference</code>	创建一个新序列，新序列中每个新值代表当前元素与上一个元素的差。重载版本用指定二元操作计算相邻元素的差

第五章 参考

<http://www.cnblogs.com/biyemyhjob/archive/2012/07/22/2603525.html>