


C++中STL用法超详细总结

 C/C++ 同时被 2 个专栏收录 ▾

10 订阅 21 篇文章

订阅专栏

目录

1 什么是STL?

2 STL内容介绍

2.1 容器

2.2 STL迭代器

2.3 算法

2.4 仿函数

2.4.1 概述

2.4.2 仿函数(functor)在编程语言中的应用

2.4.3 仿函数在STL中的定义

2.5 容器适配器

2.5.1 stack

2.5.2 queue & priority_queue

3 常用容器用法介绍

3.1 vector

3.1.1 基本函数实现

3.1.2 基本用法

3.1.3 简单介绍

3.1.4 实例

3.2 deque

3.2.1 声明deque容器

3.2.2 deque的常用成员函数

3.2.3 deque的一些特点

3.2.4 实例

3.3 list

3.3.1 list定义

3.3.2 list定义和初始化

3.3.3 list常用操作函数

3.3.4 List使用实例

3.4 map/multimap

3.4.1 基本操作函数

3.4.2 声明

3.4.3 迭代器

3.4.4 插入操作

3.4.5 查找、删除、交换

3.4.6 容量

3.4.7 排序

3.4.8 unordered_map

3.5 set/multiset

3.5.1 set常用成员函数

3.5.2 代码示例

3.5.3 unordered_set

1 什么是STL?

STL (Standard Template Library) , 即标准模板库, 是一个具有工业强度的, 高效的C++程序库。它被容纳于C++标准程序库 (C++ Standard Library) 中, 是ANSI/ISO C++标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法。为广大C++程序员们提供了一个可扩展的应用框架, 高度体现了软件的可复用性。

STL的一个重要特点是数据结构和算法的分离。尽管这是个简单的概念, 但这种分离确实使得STL变得非常通用。例如, 由于STL的sort()函数是完全通用的, 你可以用它来操作几乎任何数据集合, 包括链表, 容器和数组;

STL另一个重要特性是它不是面向对象的。为了具有足够通用性, STL主要依赖于模板而不是封装, 继承和虚函数 (多态性) ——OOP的三个要素。你在STL中找不到任何明显的类继承关系。这好像是一种倒退, 但这正好是使得STL的组件具有广泛通用性的底层特征。另外, 由于STL是基于模板, 内联函数的使用使得生成的代码短小高效;

从逻辑层次来看, 在STL中体现了泛型化程序设计思想, 引入了诸多新的名词, 比如像需求 (requirements) , 概念 (concept) , 模型 (model) , 容器 (container) , 算法 (algorithmn) , 迭代子 (iterator) 等。与OOP (object-oriented programming) 中的多态 (polymorphism) 一样, 泛型也是一种软件的复用技术;

从实现层次看, 整个STL是以一种类型参数化的方式实现的, 这种方式基于一个在早先C++标准中没有出现的语言特性--模板 (template) 。

2 STL内容介绍

STL中六大组件:

- 容器 (Container) , 是一种数据结构, 如list, vector, 和deques , 以模板类的方法提供。为了访问容器中的数据, 可以使用由容器类输出的迭代器;
- 迭代器 (Iterator) , 提供了访问容器中对象的方法。例如, 可以使用一对迭代器指定list或vector中的一些范围的对象。迭代器就如同一个指针。事实上, C++的指针也是一种迭代器。但是, 迭代器也可以是那些定义了operator*()以及其他类似于指针的操作符地方法的类对象;
- 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;
- 仿函数 (Functor)
- 适配器 (Adaptor)
- 分配器 (allocator)

2.1 容器

STL中的容器有队列容器和关联容器，容器适配器（congtainer adapters: stack,queue, priority queue），位集（bit_set），串包（string_package)等等。

（1）**序列式容器（Sequence containers）**，每个元素都有固定位置 - - 取决于插入时机和地点，和元素值无关，vector、deque、list;

Vector：将元素置于一个动态数组中加以管理，可以随机存取元素（用索引直接存取），数组尾部添加或移除元素非常快速。但是在中部或头部安插元素比较费时；

Deque：是"double-ended queue"的缩写，可以随机存取元素（用索引直接存取），数组头部和尾部添加或移除元素都非常快速。但是在中部或头部安插元素比较费时；

List：双向链表，不提供随机存取（按顺序走到需存取的元素，O(n)），在任何位置上执行插入或删除动作都非常迅速，内部只需调整一下指针；

（2）**关联式容器（Associated containers）**，元素位置取决于特定的排序准则，和插入顺序无关，set、multiset、map、multimap等。

Set/Multiset：内部的元素依据其值自动排序，Set内的相同数值的元素只能出现一次，Multisets内可包含多个数值相同的元素，内部由二叉树实现，便于查找；

Map/Multimap：Map的元素是成对的键值/实值，内部的元素依据其值自动排序，Map内的相同数值的元素只能出现一次，Multimaps内可包含多个数值相同的元素，内部由二叉树实现，便于查找；

容器类自动申请和释放内存，无需new和delete操作。

2.2 STL迭代器

Iterator（迭代器）模式又称Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。

迭代器的作用：能够让迭代器与算法不干扰的相互发展，最后又能无间隙的粘合起来，重载了*，++，==，!=，=运算符。用以操作复杂的数据结构，容器提供迭代器，算法使用迭代器；常见的一些迭代器类型：iterator、const_iterator、reverse_iterator和const_reverse_iterator。

2.3 算法

函数库对数据类型的选择对其可重用性起着至关重要的作用。举例来说，一个求方根的函数，在使用浮点数作为其参数类型的情况下的可重用性肯定比使用整型作为它的参数类性要高。而C++通过模板的机制允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化的时候，STL就利用了这一点提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的——你可以将所有的类型划分为少数的几类，然后就可以在模版的参数中使用一种类型替换掉同一种类中的其他类型。

STL提供了大约100个实现算法的模版函数，比如算法for_each将为指定序列中的每一个元素调用指定的函数，stable_sort以你所指定的规则对序列进行稳定性排序等等。只要我们熟悉了STL之后，许多代码可以被大大的化简，只需要通过调用一两个算法模板，就可以完成所需要的功能并大大地提升效率。

算法部分主要由头文件<algorithm>，<numeric>和<functional>组成。

<algorithm>是所有STL头文件中最大的一个（尽管它很好理解），它是由一大堆模版函数组成的，可以认为每个函数在很大程度上都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等等。

<numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。

<functional>中则定义了一些模板类，用以声明函数对象。

STL中算法大致分为四类：

- 非可变序列算法：指不直接修改其所操作的容器内容的算法。
- 可变序列算法：指可以修改它们所操作的容器内容的算法。
- 排序算法：对序列进行排序和合并的算法、搜索算法以及有序序列上的集合操作。
- 数值算法：对容器内容进行数值计算。

以下对所有算法进行细致分类并标明功能：

<==>查找算法(13个)：判断容器中是否包含某个值

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。重载版本使用输入的二元操作符代替相等的判断。

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。

<二>排序和通用算法(14个)：提供元素排序策略

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n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。

<三>删除和替换算法(15个)

copy：复制序列

copy_backward：与copy相同，不过元素是以相反顺序被拷贝。

iter_swap：交换两个ForwardIterator的值。

remove：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remove和remove_if函数。

remove_copy：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。

remove_if：删除指定范围内输入操作结果为true的所有元素。

remove_copy_if：将所有不匹配元素拷贝到一个指定容器。

replace：将指定范围内所有等于vold的元素都用vnew代替。

replace_copy：与replace类似，不过将结果写入另一个容器。

replace_if：将指定范围内所有操作结果为true的元素用新值代替。

replace_copy_if：与replace_if，不过将结果写入另一个容器。

swap：交换存储在两个对象中的值。

swap_range：将指定范围内的元素与另一个序列元素值进行交换。

unique：清除序列中重复元素，和remove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。

unique_copy：与unique类似，不过把结果输出到另一个容器。

<四>排列组合算法(2个)：提供计算给定集合按一定顺序的所有可能排列组合

next_permutation：取出当前范围内的排列，并重新排序为下一个排列。重载版本使用自定义的比较操作。

prev_permutation：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false。重载版本使用自定义的比较操作。

<五>算术算法(4个)

accumulate：iterator对标识的序列段元素之和，加到一个由val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。

partial_sum：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。

inner_product：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。

adjacent_difference：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。

<六>生成和异变算法(6个)

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。

<七>关系算法(8个)

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ForwardIterator，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ForwardIterator，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。

<八>集合算法(4个)

set_union: 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。
set_intersection: 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。
set_difference: 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。

set_symmetric_difference: 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)。

<九>堆算法(4个)

make_heap: 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。
pop_heap: 并不真正把最大元素从堆中弹出，而是重新排序堆。它把first和last-1交换，然后重新生成一个堆。可使用容器的back来访问被”弹出”的元素或者使用pop_back进行真正的删除。重载版本使用自定义的比较操作。
push_heap: 假设first到last-1是一个有效堆，要被加入到堆的元素存放在位置last-1，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。
sort_heap: 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。

2.4 仿函数

2.4.1 概述

仿函数(functor)，就是使一个类的使用看上去象一个函数。其实实现就是类中实现一个operator()，这个类就有了类似函数的行为，就是一个仿函数类了。
有些功能的的代码，会在不同的成员函数中用到，想复用这些代码。

1) 公共的函数，可以，这是一个解决方法，不过函数用到的一些变量，就可能成为公共的全局变量，再说为了复用这么一片代码，就要单立出一个函数，也不是很好维护。

2) 仿函数，写一个简单类，除了那些维护一个类的成员函数外，就只是实现一个operator()，在类实例化时，就将要用的，非参数的元素传入类中。

2.4.2 仿函数(functor)在编程语言中的应用

1) C语言使用**函数指针**和**回调函数**来实现仿函数，例如一个用来排序的函数可以这样使用仿函数

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 //int sort_function( const void *a, const void *b);
4 int sort_function( const void *a, const void *b)
5 {
6     return *(int*)a-*(int*)b;
7 }
8
9 int main()
10 {
11
12     int list[5] = { 54, 21, 11, 67, 22 };
13     qsort((void *)list, 5, sizeof(list[0]), sort_function);//起始地址，个数，元素大小，回调函数
14     int x;
15     for (x = 0; x < 5; x++)
16         printf("%i\n", list[x]);
17
18     return 0;
19 }
```

2) 在C++里，我们通过在一个类中重载括号运算符的方法使用一个函数对象而不是一个普通函数。

```
1 #include <iostream>
2 #include <algorithm>
3
4 using namespace std;
5 template<typename T>
6 class display
7 {
8 public:
9     void operator()(const T &x)
10     {
11         cout << x << " ";
12     }
13 };
14 int main()
15 {
16     int ia[] = { 1,2,3,4,5 };
17     for_each(ia, ia + 5, display<int>());
18     system("pause");
19     return 0;
20 }
```

2.4.3 仿函数在STL中的定义

要使用STL内建的仿函数，必须包含<functional>头文件。而头文件中包含的仿函数分类包括

1) 算术类仿函数

- 加: plus<T>
- 减: minus<T>
- 乘: multiplies<T>
- 除: divides<T>
- 模取: modulus<T>
- 否定: negate<T>

例子:

```
1 #include <iostream>
2 #include <numeric>
3 #include <vector>
4 #include <functional>
5 using namespace std;
6
7 int main()
8 {
9     int ia[] = { 1,2,3,4,5 };
10    vector<int> iv(ia, ia + 5);
11    //120
12    cout << accumulate(iv.begin(), iv.end(), 1, multiplies<int>()) << endl;
13    //15
14    cout << multiplies<int>()(3, 5) << endl;
15
16    modulus<int> modulusObj;
17    cout << modulusObj(3, 5) << endl; // 3
18    system("pause");
19    return 0;
20 }
```

20 | }

2) 关系运算类仿函数

等于: equal_to<T>

不等于: not_equal_to<T>

大于: greater<T>

大于等于: greater_equal<T>

小于: less<T>

小于等于: less_equal<T>

从大到小排序:

```
1 #include <iostream>
2 #include <algorithm>
3 #include<functional>
4 #include <vector>
5
6 using namespace std;
7
8 template <class T>
9 class display
10 {
11 public:
12     void operator()(const T &x)
13     {
14         cout << x << " ";
15     }
16 };
17 int main()
18 {
19     int ia[] = { 1,5,4,3,2 };
20     vector<int> iv(ia, ia + 5);
21     sort(iv.begin(), iv.end(), greater<int>());
22     for_each(iv.begin(), iv.end(), display<int>());
23     system("pause");
24     return 0;
25 }
```

3) 逻辑运算仿函数

逻辑与: logical_and<T>

逻辑或: logical_or<T>

逻辑否: logical_no<T>

除了使用STL内建的仿函数，还可使用自定义的仿函数，具体实例见文章3.4.7.2小结

2.5 容器适配器

标准库提供了三种顺序容器适配器：queue(FIFO队列)、priority_queue(优先级队列)、stack(栈)

- 什么是容器适配器

"适配器是使一种事物的行为类似于另外一种事物行为的一种机制"，适配器对容器进行包装，使其表现出另外一种行为。例如，stack<int, vector<int> >实现了栈的功能，但其内部使用顺序容器vector<int>来存储数据。（相当于是vector<int>表现出 了栈的行为）。

- 容器适配器

要使用适配器，需要加入一下头文件：

```
#include <stack>           //stack

#include<queue>           //queue、priority_queue
```

种类	默认顺序容器	可用顺序容器	说明
stack	deque	vector、list、deque	
queue	deque	list、deque	基础容器必须提供push_front()运算
priority_queue	vector	vector、deque	基础容器必须提供随机访问功能

- 定义适配器

1、初始化

```
stack<int> stk(dep);
```

2、覆盖默认容器类型

```
stack<int,vector<int> > stk;
```

- 使用适配器

2.5.1 stack

```
1 stack<int> s;
2 stack< int, vector<int> > stk; //覆盖基础容器类型，使用vector实现stk
3 s.empty(); //判断stack是否为空，为空返回true，否则返回false
4 s.size(); //返回stack中元素的个数
5 s.pop(); //删除栈顶元素，但不返回其值
6 s.top(); //返回栈顶元素的值，但不删除此元素
7 s.push(item); //在栈顶压入新元素item
```

实例：括号匹配

```
1 #include<iostream>
2 #include<cstdio>
3 #include<string>
4 #include<stack>
5 using namespace std;
6 int main()
7 {
8     string s;
9     stack<char> ss;
10    while (cin >> s)
11    {
12        bool flag = true;
13        for (char c : s) //C++11新标准，即遍历一次字符串s
14        {
15            if (c == '(' || c == '{' || c == '[')
16            {
17                ss.push(c);
18                continue;
19            }
20            if (c == '}')
21            {
22                if (!ss.empty() && ss.top() == '{')
23                {
24                    ss.pop();
25                    continue;
26                }
27                else
28                {
29                    flag = false;
30                    break;
31                }
32            }
```

```
33         if (!ss.empty() && c == ']')34         {
35             if (ss.top() == '[')
36             {
37                 ss.pop();
38                 continue;
39             }
40             else
41             {
42                 flag = false;
43                 break;
44             }
45         }
46         if (!ss.empty() && c == ')')
47         {
48             if (ss.top() == '(')
49             {
50                 ss.pop();
51                 continue;
52             }
53             else
54             {
55                 flag = false;
56                 break;
57             }
58         }
59     }
60     if (flag)    cout << "Match!" << endl;
61     else        cout << "Not Match!" << endl;
62 }
63 }
```

2.5.2 queue & priority_queue

```
1 queue<int> q; //priority_queue<int> q;
2 q.empty(); //判断队列是否为空
3 q.size(); //返回队列长度
4 q.push(item); //对于queue, 在队尾压入一个新元素
5 //对于priority_queue, 在基于优先级的适当位置插入新元素
6
7 //queue only:
8 q.front(); //返回队首元素的值, 但不删除该元素
9 q.back(); //返回队尾元素的值, 但不删除该元素
10
11 //priority_queue only:
12 q.top(); //返回具有最高优先级的元素值, 但不删除该元素
```

3 常用容器用法介绍

3.1 vector

3.1.1 基本函数实现

1.构造函数

- vector():创建一个空vector
- vector(int nSize):创建一个vector,元素个数为nSize
- vector(int nSize,const t& t):创建一个vector, 元素个数为nSize,且值均为t
- vector(const vector&):复制构造函数
- vector(begin,end):复制[begin,end)区间内另一个数组的元素到vector中

2.增加函数

- void push_back(const T& x):向量尾部增加一个元素X
- iterator insert(iterator it,const T& x):向量中迭代器指向元素前增加一个元素x
- iterator insert(iterator it,int n,const T& x):向量中迭代器指向元素前增加n个相同的元素x
- iterator insert(iterator it,const_iterator first,const_iterator last):向量中迭代器指向元素前插入另一个相同类型向量的[first,last)间的数据

3.删除函数

- iterator erase(iterator it):删除向量中迭代器指向元素
- iterator erase(iterator first,iterator last):删除向量中[first,last)中元素
- void pop_back():删除向量中最后一个元素
- void clear():清空向量中所有元素

4.遍历函数

- reference at(int pos):返回pos位置元素的引用
- reference front():返回首元素的引用
- reference back():返回尾元素的引用
- iterator begin():返回向量头指针, 指向第一个元素
- iterator end():返回向量尾指针, 指向向量最后一个元素的下一个位置
- reverse_iterator rbegin():反向迭代器, 指向最后一个元素
- reverse_iterator rend():反向迭代器, 指向第一个元素之前的位置

5.判断函数

- bool empty() const:判断向量是否为空, 若为空, 则向量中无元素

6.大小函数

- int size() const:返回向量中元素的个数
- int capacity() const:返回当前向量张红所能容纳的最大元素值
- int max_size() const:返回最大可允许的vector元素数量值

7.其他函数

- void swap(vector&):交换两个同类型向量的数据
- void assign(int n,const T& x):设置向量中第n个元素的值为x
- void assign(const_iterator first,const_iterator last):向量中[first,last)中元素设置成当前向量元素

8.看着清楚

- 1.push_back 在数组的最后添加一个数据
- 2.pop_back 去掉数组的最后一个数据
- 3.at 得到编号位置的数据
- 4.begin 得到数组头的指针
- 5.end 得到数组的最后一个单元+1的指针
6. front 得到数组头的引用
- 7.back 得到数组的最后一个单元的引用
- 8.max_size 得到vector最大可以是多大
- 9.capacity 当前vector分配的大小
- 10.size 当前使用数据的大小
- 11.resize 改变当前使用数据的大小, 如果它比当前使用的大, 者填充默认值
- 12.reserve 改变当前vecotr所分配空间的大小

- 13.erase 删除指针指向的数据项
- 14.clear 清空当前的vector
- 15.rbegin 将vector反转后的开始指针返回(其实就是原来的end-1)
- 16.rend 将vector反转构的结束指针返回(其实就是原来的begin-1)
- 17.empty 判断vector是否为空
- 18.swap 与另一个vector交换数据

3.1.2 基本用法

```
1 | #include < vector>
2 | using namespace std;
```

3.1.3 简单介绍

1. Vector<类型>标识符
2. Vector<类型>标识符(最大容量)
3. Vector<类型>标识符(最大容量,初始所有值)
4. Int i[5]={1,2,3,4,5}
Vector<类型>vi(l,i+2)//得到索引值为3以后的值
5. Vector< vector< int> >v; 二维向量/这里最外的<>要有空格。否则在比较旧的编译器下无法通过

3.1.4 实例

3.1.4.1 pop_back()&push_back(elem)实例在容器最后移除和插入数据

```
1 | #include <string.h>
2 | #include <vector>
3 | #include <iostream>
4 | using namespace std;
5 |
6 | int main()
7 | {
8 |     vector<int>obj;//创建一个向量存储容器 int
9 |     for(int i=0;i<10;i++) // push_back(elem) 在数组最后添加数据
10 |    {
11 |        obj.push_back(i);
12 |        cout<<obj[i]<<" ";
13 |    }
14 |
15 |    for(int i=0;i<5;i++)//去掉数组最后一个数据
16 |    {
17 |        obj.pop_back();
18 |    }
19 |
20 |    cout<<"\n"<<endl;
21 |
22 |    for(int i=0;i<obj.size();i++)//size() 容器中实际数据个数
23 |    {
24 |        cout<<obj[i]<<" ";
25 |    }
26 |
27 |    return 0;
28 | }
```

输出结果为:

```
1 | 0,1,2,3,4,5,6,7,8,9,
2 |
3 | 0,1,2,3,4,
```

3.1.4.2 clear()清除容器中所有数据

```
1 | #include <string.h>
2 | #include <vector>
3 | #include <iostream>
4 | using namespace std;
5 |
6 | int main()
7 | {
8 |     vector<int>obj;
9 |     for(int i=0;i<10;i++)//push_back(elem) 在数组最后添加数据
10 |    {
11 |        obj.push_back(i);
12 |        cout<<obj[i]<<" ";
13 |    }
14 |
15 |    obj.clear();//清除容器中所以数据
16 |    for(int i=0;i<obj.size();i++)
17 |    {
18 |        cout<<obj[i]<<endl;
19 |    }
20 |
21 |    return 0;
22 | }
```

输出结果为:

```
0,1,2,3,4,5,6,7,8,9,
```

3.1.4.3 排序

```
1 | #include <string.h>
2 | #include <vector>
3 | #include <iostream>
4 | #include <algorithm>
5 | using namespace std;
6 |
7 | int main()
8 | {
9 |     vector<int>obj;
10 |
11 |     obj.push_back(1);
12 |     obj.push_back(3);
13 |     obj.push_back(0);
14 |
15 |     sort(obj.begin(),obj.end());//从小到大
16 |
17 |     cout<<"从小到大:"<<endl;
18 |     for(int i=0;i<obj.size();i++)
19 |     {
20 |         cout<<obj[i]<<" ";
21 |     }
22 |
23 |     cout<<"\n"<<endl;
24 |
25 |     cout<<"从大到小:"<<endl;
26 |     reverse(obj.begin(),obj.end());//从大到小
27 |     for(int i=0;i<obj.size();i++)
28 |     {
29 |         cout<<obj[i]<<" ";
30 |     }
31 |     return 0;
32 | }
```

输出结果为：

```
1 | 从小到大：
2 | 0,1,3,
3 |
4 | 从大到小：
5 | 3,1,0,
```

1.注意 sort 需要头文件 #include <algorithm>

2.如果想 sort 来降序， 可重写 sort

```
1 | bool compare(int a,int b)
2 | {
3 |     return a< b; //升序排列， 如果改为return a>b, 则为降序
4 | }
5 | int a[20]={2,4,1,23,5,76,0,43,24,65},i;
6 | for(i=0;i<20;i++)
7 |     cout<< a[i]<< endl;
8 | sort(a,a+20,compare);
```

3.1.4.4 访问（直接数组访问&迭代器访问）

```
1 | #include <string.h>
2 | #include <vector>
3 | #include <iostream>
4 | #include <algorithm>
5 | using namespace std;
6 |
7 | int main()
8 | {
9 |     //顺序访问
10 |    vector<int>obj;
11 |    for(int i=0;i<10;i++)
12 |    {
13 |        obj.push_back(i);
14 |    }
15 |
16 |    cout<<"直接利用数组： ";
17 |    for(int i=0;i<10;i++)//方法一
18 |    {
19 |        cout<<obj[i]<<" ";
20 |    }
21 |
22 |    cout<<endl;
23 |    cout<<"利用迭代器： " ;
24 |    //方法二， 使用迭代器将容器中数据输出
25 |    vector<int>::iterator it;//声明一个迭代器， 来访问vector容器， 作用： 遍历或者指向vector容器的元素
26 |    for(it=obj.begin();it!=obj.end();it++)
27 |    {
28 |        cout<<*it<<" ";
29 |    }
30 |    return 0;
31 | }
```

输出结果为：

```
1 | 直接利用数组： 0 1 2 3 4 5 6 7 8 9
2 | 利用迭代器： 0 1 2 3 4 5 6 7 8 9
```

3.1.4.5 二维数组两种定义方法（结果一样）

方法一

```
1 | #include <string.h>
2 | #include <vector>
3 | #include <iostream>
4 | #include <algorithm>
5 | using namespace std;
6 |
7 |
8 | int main()
9 | {
10 |    int N=5, M=6;
11 |    vector<vector<int> > obj(N); //定义二维动态数组大小5行
12 |    for(int i =0; i< obj.size(); i++)//动态二维数组为5行6列， 值全为0
13 |    {
14 |        obj[i].resize(M);
15 |    }
16 |
17 |    for(int i=0; i< obj.size(); i++)//输出二维动态数组
18 |    {
19 |        for(int j=0;j<obj[i].size();j++)
20 |        {
21 |            cout<<obj[i][j]<<" ";
22 |        }
23 |        cout<<"\n";
24 |    }
25 |    return 0;
26 | }
```

方法二

```
1 | #include <string.h>
2 | #include <vector>
3 | #include <iostream>
4 | #include <algorithm>
5 | using namespace std;
6 |
7 |
8 | int main()
9 | {
10 |    int N=5, M=6;
11 |    vector<vector<int> > obj(N, vector<int>(M)); //定义二维动态数组5行6列
12 |
13 |    for(int i=0; i< obj.size(); i++)//输出二维动态数组
14 |    {
15 |        for(int j=0;j<obj[i].size();j++)
16 |        {
17 |            cout<<obj[i][j]<<" ";
18 |        }
19 |        cout<<"\n";
20 |    }
21 |    return 0;
22 | }
```

输出结果为：

```
1 | 0 0 0 0 0 0
2 | 0 0 0 0 0 0
3 | 0 0 0 0 0 0
4 | 0 0 0 0 0 0
5 | 0 0 0 0 0 0
```

3.2 deque

所谓的deque是”double ended queue”的缩写， 双端队列不论在尾部或头部插入元素， 都十分迅速。而在中间插入元素则会比较费时， 因为必须移动中间其他的元素。双端队列是一种随机访问的数据类型， 提供了在序列两端快速插入和删除操作的功能， 它可以在需要的时候改变自身大小， 完成了标准的C++数据结构中队列的所有功能。

Vector是单向开口的连续线性空间，deque则是一种双向开口的连续线性空间。deque对象在队列的两端放置元素和删除元素是高效的，而向量vector只是在插入序列的末尾时操作才是高效的。deque和vector的最大差异，一在于deque允许于常数时间内对头端进行元素的插入或移除操作，二在于deque没有所谓的capacity观念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来。换句话说，像vector那样“因旧空间不足而重新配置一块更大空间，然后复制元素，再释放旧空间”这样的事情在deque中是不会发生的。也因此，deque没有必要提供所谓的空间预留（reserved）功能。

虽然deque也提供Random Access Iterator，但它的迭代器并不是普通指针，其复杂度和vector不可同日而语，这当然涉及到各个运算层面。因此，除非必要，我们应尽可能选择使用vector而非deque。对deque进行的排序操作，为了最高效率，可将deque先完整复制到一个vector身上，将vector排序后（利用STL的sort算法），再复制回deque。

deque是一种优化了的对序列两端元素进行添加和删除操作的基本序列容器。通常由一些独立的区块组成，第一区块朝某方向扩展，最后一个区块朝另一方向扩展。它允许较为快速地随机访问但它不像vector一样把所有对象保存在一个连续的内存块，而是多个连续的内存块。并且在一个映射结构中保存对这些块以及顺序的跟踪。

3.2.1 声明deque容器

```
1 | #include<deque> // 头文件
2 | deque<type> deq; // 声明一个元素类型为type的双端队列deq
3 | deque<type> deq(size); // 声明一个类型为type、含有size个默认值初始化元素的的双端队列deq
4 | deque<type> deq(size, value); // 声明一个元素类型为type、含有size个value元素的双端队列deq
5 | deque<type> deq(mydeque); // deq是mydeque的一个副本
6 | deque<type> deq(first, last); // 使用迭代器first、last范围内的元素初始化deq
```

3.2.2 deque的常用成员函数

```
deque<int> deq;
```

- deq[]：用来访问双向队列中单个的元素。
- deq.front()：返回第一个元素的引用。
- deq.back()：返回最后一个元素的引用。
- deq.push_front(x)：把元素x插入到双向队列的头部。
- deq.pop_front()：弹出双向队列的第一个元素。
- deq.push_back(x)：把元素x插入到双向队列的尾部。
- deq.pop_back()：弹出双向队列的最后一个元素。

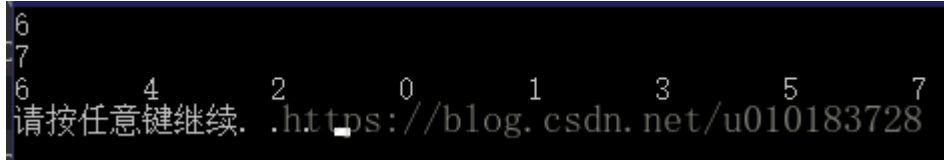
3.2.3 deque的一些特点

- 支持随机访问，即支持[]以及at()，但是性能没有vector好。
- 可以在内部进行插入和删除操作，但性能不及list。
- deque两端都能够快速插入和删除元素，而vector只能在尾端进行。
- deque的元素存取和迭代器操作会稍微慢一些，因为deque的内部结构会多一个间接过程。
- deque迭代器是特殊的智能指针，而不是一般指针，它需要在不同的区块之间跳转。
- deque可以包含更多的元素，其max_size可能更大，因为不止使用一块内存。
- deque不支持对容量和内存分配时机的控制。
- 在除了首尾两端的其他地方插入和删除元素，都将会导致指向deque元素的任何pointers、references、iterators失效。不过，deque的内存重分配优于vector，因为其内部结构显示不需要复制所有元素。
- deque的内存区块不再被使用时，会被释放，deque的内存大小是可缩减的。不过，是不是这么做以及怎么做由实际操作版本定义。
- deque不提供容量操作：capacity()和reverse()，但是vector可以。

3.2.4 实例

```
1 | #include<iostream>
2 | #include<stdio.h>
3 | #include<deque>
4 | using namespace std;
5 | int main(void)
6 | {
7 |     int i;
8 |     int a[10] = { 0,1,2,3,4,5,6,7,8,9 };
9 |     deque<int> q;
10 |    for (i = 0; i <= 9; i++)
11 |    {
12 |        if (i % 2 == 0)
13 |            q.push_front(a[i]);
14 |        else
15 |            q.push_back(a[i]);
16 |    } //此时队列里的内容是: {8,6,4,2,0,1,3,5,7,9}*/
17 |    q.pop_front();
18 |    printf("%d\n", q.front()); //清除第一个元素后输出第一个(6)*/
19 |    q.pop_back();
20 |    printf("%d\n", q.back()); //清除最后一个元素后输出最后一个(7)*/
21 |    deque<int>::iterator it;
22 |    for (it = q.begin(); it != q.end(); it++) {
23 |        cout << *it << '\t';
24 |    }
25 |    cout << endl;
26 |    system("pause");
27 |    return 0;
28 | }
```

输出结果：



3.3 list

3.3.1 list定义

List是stl实现的双向链表，与向量(vectors)相比，它允许快速的插入和删除，但是随机访问却比较慢。使用时需要添加头文件

```
#include <list>
```

3.3.2 list定义和初始化

```
list<int> lst1; //创建空list

list<int> lst2(5); //创建含有5个元素的list

list<int> lst3(3,2); //创建含有3个元素的list

list<int> lst4(lst2); //使用lst2初始化lst4

list<int> lst5(lst2.begin(),lst2.end()); //同lst4
```

3.3.3 list常用操作函数

Lst1.assign() 给lis赋值
Lst1.back() 返回最后一个元素
Lst1.begin() 返回指向第一个元素的迭代器
Lst1.clear() 删除所有元素
Lst1.empty() 如果list是空的则返回true
Lst1.end() 返回末尾的迭代器
Lst1.erase() 删除一个元素
Lst1.front() 返回第一个元素
Lst1.get_allocator() 返回list的配置器
Lst1.insert() 插入一个元素到list中
Lst1.max_size() 返回list能容纳的最大元素数量
Lst1.merge() 合并两个list
Lst1.pop_back() 删除最后一个元素
Lst1.pop_front() 删除第一个元素
Lst1.push_back() 在list的末尾添加一个元素
Lst1.push_front() 在list的头部添加一个元素

Lst1.rbegin() 返回指向第一个元素的逆向迭代器
Lst1.remove() 从list删除元素
Lst1.remove_if() 按指定条件删除元素
Lst1.rend() 指向list末尾的逆向迭代器
Lst1.resize() 改变list的大小
Lst1.reverse() 把list的元素倒转
Lst1.size() 返回list中的元素个数
Lst1.sort() 给list排序
Lst1.splice() 合并两个list
Lst1.swap() 交换两个list
Lst1.unique() 删除list中相邻重复的元素

3.3.4 List使用实例

3.3.4.1 迭代器遍历list

```
1   for(list<int>::const_iterator iter = lst1.begin(); iter != lst1.end(); iter++)
2   {
3       cout<<*iter;
4   }
5   cout<<endl;
```

3.3.4.2 综合实例1

```
1   #include <iostream>
2   #include <list>
3   #include <numeric>
4   #include <algorithm>
5   using namespace std;
6
7   typedef list<int> LISTINT;
8   typedef list<int> LISTCHAR;
9
10  void main()
11  {
12      //用LISTINT 创建一个List对象
13      LISTINT listOne;
14      //声明i为迭代器
15      LISTINT::iterator i;
16
17      listOne.push_front(3);
18      listOne.push_front(2);
19      listOne.push_front(1);
20
21      listOne.push_back(4);
22      listOne.push_back(5);
23      listOne.push_back(6);
24
25      cout << "listOne.begin()--- listOne.end():" << endl;
26      for (i = listOne.begin(); i != listOne.end(); ++i)
27          cout << *i << " ";
28      cout << endl;
29
30      LISTINT::reverse_iterator ir;
31      cout << "listOne.rbegin()---listOne.rend():" << endl;
32      for (ir = listOne.rbegin(); ir != listOne.rend(); ir++) {
33          cout << *ir << " ";
34      }
35      cout << endl;
36
37      int result = accumulate(listOne.begin(), listOne.end(), 0);
38      cout << "Sum=" << result << endl;
39      cout << "-----" << endl;
40
41      //用LISTCHAR 创建一个List对象
42      LISTCHAR listTwo;
43      //声明j为迭代器
44      LISTCHAR::iterator j;
45
46      listTwo.push_front('C');
47      listTwo.push_front('B');
48      listTwo.push_front('A');
49
50      listTwo.push_back('D');
51      listTwo.push_back('E');
52      listTwo.push_back('F');
53
54      cout << "listTwo.begin()---listTwo.end():" << endl;
55      for (j = listTwo.begin(); j != listTwo.end(); ++j)
56          cout << char(*j) << " ";
57      cout << endl;
58
59      j = max_element(listTwo.begin(), listTwo.end());
60      cout << "The maximum element in listTwo is: " << char(*j) << endl;
61      system("pause");
62  }
```

输出结果

```
listOne.begin()--- listOne.end():
1 2 3 4 5 6
listOne.rbegin()---listOne.rend():
6 5 4 3 2 1
Sum=21
-----
listTwo.begin()---listTwo.end():
A B C D E F
The maximum element in listTwo is: F
请按任意键继续. . .
```

3.3.4.3 综合实例2

```
1   #include <iostream>
2   #include <list>
3
4   using namespace std;
5   typedef list<int> INTLIST;
6
7   //从前向后显示List队列的全部元素
8   void put_list(INTLIST list, char *name)
9   {
10      INTLIST::iterator plist;
11
12      cout << "The contents of " << name << " : ";
13      for (plist = list.begin(); plist != list.end(); plist++)
14          cout << *plist << " ";
15      cout << endl;
16  }
17
18  //测试list容器的功能
19  void main(void)
20  {
21      //list1对象初始为空
22      INTLIST list1;
23      INTLIST list2(5, 1);
24      INTLIST list3(list2.begin(), --list2.end());
25
26      //声明一个名为i的双向迭代器
27      INTLIST::iterator i;
28
29      put_list(list1, "list1");
30      put_list(list2, "list2");
31      put_list(list3, "list3");
32
33      list1.push_back(7);
34      list1.push_back(8);
35      cout << "list1.push_back(7) and list1.push_back(8):" << endl;
```

```

37 | put_list(list1, "list1");
38 | list1.push_front(6);
39 | list1.push_front(5);
40 | cout << "list1.push_front(6) and list1.push_front(5):" << endl;
41 | put_list(list1, "list1");
42 |
43 | list1.insert(++list1.begin(), 3, 9);
44 | cout << "list1.insert(list1.begin()+1,3,9):" << endl;
45 | put_list(list1, "list1");
46 |
47 | //测试引用类函数
48 | cout << "list1.front()=" << list1.front() << endl;
49 | cout << "list1.back()=" << list1.back() << endl;
50 |
51 | list1.pop_front();
52 | list1.pop_back();
53 | cout << "list1.pop_front() and list1.pop_back():" << endl;
54 | put_list(list1, "list1");
55 |
56 | list1.erase(++list1.begin());
57 | cout << "list1.erase(++list1.begin()):" << endl;
58 | put_list(list1, "list1");
59 |
60 | list2.assign(8, 1);
61 | cout << "list2.assign(8,1):" << endl;
62 | put_list(list2, "list2");
63 |
64 | cout << "list1.max_size(): " << list1.max_size() << endl;
65 | cout << "list1.size(): " << list1.size() << endl;
66 | cout << "list1.empty(): " << list1.empty() << endl;
67 |
68 | put_list(list1, "list1");
69 | put_list(list3, "list3");
70 | cout << "list1>list3: " << (list1 > list3) << endl;
71 | cout << "list1<list3: " << (list1 < list3) << endl;
72 |
73 | list1.sort();
74 | put_list(list1, "list1");
75 |
76 | list1.splice(++list1.begin(), list3);
77 | put_list(list1, "list1");
78 | put_list(list3, "list3");
79 | system("pause");
80 | }

```

输出结果：

```

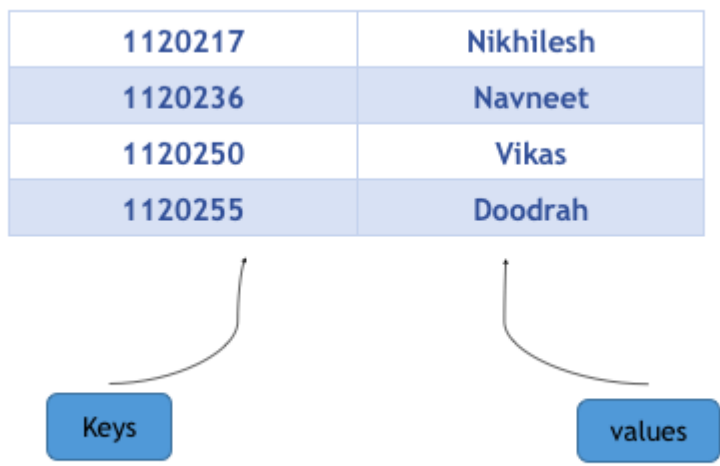
The contents of list1:
The contents of list2: 1 1 1 1 1
The contents of list3: 1 1 1 1
list1.push_back(7) and list1.push_back(8):
The contents of list1: 7 8
list1.push_front(6) and list1.push_front(5):
The contents of list1: 5 6 7 8
list1.insert(list1.begin()+1,3,9):
The contents of list1: 5 9 9 9 6 7 8
list1.front()=5
list1.back()=8
list1.pop_front() and list1.pop_back():
The contents of list1: 9 9 9 6 7
list1.erase(4,list1.begin()):
The contents of list1: 9 9 6 7
list2.assign(8,1):
The contents of list2: 1 1 1 1 1 1 1 1
list1.max_size(): 357913941
list1.size(): 4
list1.empty(): 0
The contents of list1: 9 9 6 7
The contents of list3: 1 1 1 1
list4=list3: 1
list1=list3: 0
The contents of list1: 6 7 9 9
The contents of list1: 6 7 9 9
The contents of list3:

```

3.4 map/multimap

map和multimap都需要#include<map>, 唯一的不同是, map的键值key不可重复, 而multimap可以, 也正是由于这种区别, map支持[]运算符, multimap不支持[]运算符。在用法上没什么区别。

C++中map提供的是一种键值对容器，里面的数据都是成对出现的,如下图：每一对中的第一个值称之为关键字(key)，每个关键字只能在map中出现一次；第二个称之为该关键字的对应值。



Map是STL的一个关联容器，它提供一对一（其中第一个可以称为关键字，每个关键字只能在map中出现一次，第二个可能称为该关键字的值）的数据处理能力，由于这个特性，它完成有可能在我们处理一对一数据的时候，在编程上提供快速通道。这里说下map内部数据的组织，map内部自建一颗红黑树（一种非严格意义上的平衡二叉树），这颗树具有对数据自动排序的功能，所以在map内部所有的数据都是有序的。

3.4.1 基本操作函数

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
empty()	如果map为空则返回true
end()	返回指向map末尾的迭代器
equal_range()	返回特殊条目的迭代器对
erase()	删除一个元素
find()	查找一个元素
get_allocator()	返回map的配置器
insert()	插入元素
key_comp()	返回比较元素key的函数
lower_bound()	返回键值>=给定元素的第一个位置
max_size()	返回可以容纳的最大元素个数
rbegin()	返回一个指向map尾部的逆向迭代器
rend()	返回一个指向map头部的逆向迭代器
size()	返回map中元素的个数
swap()	交换两个map
upper_bound()	返回键值>给定元素的第一个位置
value_comp()	返回比较元素value的函数

3.4.2 声明

```
1 //头文件
2 #include<map>
3
4 map<int, string> ID_Name;
```



```
5
6 // 使用{}赋值是从c++11开始的，因此编译器版本过低时会报错，如visual studio 2012
7 map<int, string> ID_Name = {
8     { 2015, "Jim" },
9     { 2016, "Tom" },
10    { 2017, "Bob" } };
```

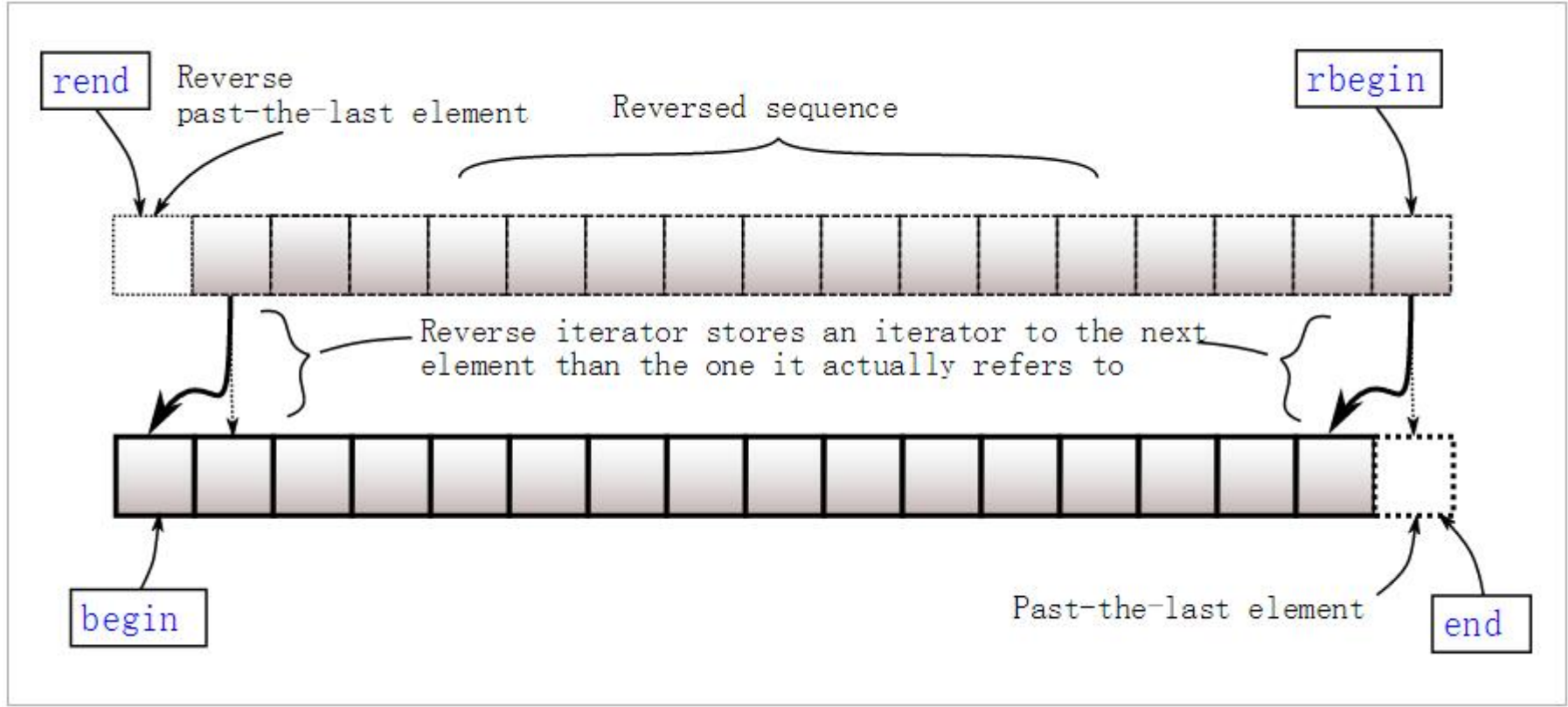
3.4.3 迭代器

共有八个获取迭代器的函数: **begin, end, rbegin, rend** 以及对应的 **cbegin, cend, crbegin, crend**。

二者的区别在于，后者一定返回 const_iterator，而前者则根据map的类型返回iterator 或者 const_iterator。const情况下，不允许对值进行修改。如下面代码所示：

```
1 map<int,int>::iterator it;
2 map<int,int> mmap;
3 const map<int,int> const_mmap;
4
5 it = mmap.begin(); //iterator
6 mmap.cbegin(); //const_iterator
7
8 const_mmap.begin(); //const_iterator
9 const_mmap.cbegin(); //const_iterator
```

返回的迭代器可以进行加减操作，此外，如果map为空，则 begin = end。



3.4.4 插入操作

3.4.4.1 用insert插入pair数据

```
1 // 数据的插入-- 第一种: 用insert函数插入pair数据
2 #include <map>
3 #include <string>
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     map<int, string> mapStudent;
10    mapStudent.insert(pair<int, string>(1, "student_one"));
11    mapStudent.insert(pair<int, string>(2, "student_two"));
12    mapStudent.insert(pair<int, string>(3, "student_three"));
13    map<int, string>::iterator iter;
14    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
15        cout<<iter->first<<' '<<iter->second<<endl;
16 }
```

3.4.4.2 用insert函数插入value_type数据

```
1 // 第二种: 用insert函数插入value_type数据，下面举例说明
2
3 #include <map>
4 #include <string>
5 #include <iostream>
6 using namespace std;
7
8 int main()
9 {
10    map<int, string> mapStudent;
11    mapStudent.insert(map<int, string>::value_type (1, "student_one"));
12    mapStudent.insert(map<int, string>::value_type (2, "student_two"));
13    mapStudent.insert(map<int, string>::value_type (3, "student_three"));
14    map<int, string>::iterator iter;
15    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
16        cout<<iter->first<<' '<<iter->second<<endl;
17 }
```

3.4.4.3 用insert函数进行多个插入

insert共有4个重载函数：

```
1 // 插入单个键值对，并返回插入位置 and 成功标志，插入位置已经存在值时，插入失败
2 pair<iterator,bool> insert (const value_type& val);
3
4 // 在指定位置插入，在不同位置插入效率是不一样的，因为涉及到重排
5 iterator insert (const_iterator position, const value_type& val);
6
7 // 插入多个
8 void insert (InputIterator first, InputIterator last);
9
10 //c++11开始支持，使用列表插入多个
11 void insert (initializer_list<value_type> il);
```

下面是具体使用示例：

```
1 #include <iostream>
2 #include <map>
3
4 int main()
5 {
6     std::map<char, int> mymap;
7
8     // 插入单个值
9     mymap.insert(std::pair<char, int>('a', 100));
10    mymap.insert(std::pair<char, int>('z', 200));
11
12    // 返回插入位置以及是否插入成功
13    std::pair<std::map<char, int>::iterator, bool> ret;
14    ret = mymap.insert(std::pair<char, int>('z', 500));
15    if (ret.second == false) {
16        std::cout << "element 'z' already existed";
17        std::cout << " with a value of " << ret.first->second << '\n';
18    }
19
20    // 指定位置插入
21    std::map<char, int>::iterator it = mymap.begin();
22    mymap.insert(it, std::pair<char, int>('b', 300)); // 效率更高
23    mymap.insert(it, std::pair<char, int>('c', 400)); // 效率非最高
24
25    // 范围多值插入
26    std::map<char, int> anothermap;
27    anothermap.insert(mymap.begin(), mymap.find('c'));
28
29    // 列表形式插入
```

```
30 |         anothermap.insert({ { 'd', 100 }, { 'e', 200 } });
31 |
32 |     return 0;
33 | }
```

3.4.4.4 用数组方式插入数据

```
1 | // 第三种：用数组方式插入数据，下面举例说明
2 |
3 | #include <map>
4 | #include <string>
5 | #include <iostream>
6 | using namespace std;
7 |
8 | int main()
9 | {
10 |     map<int, string> mapStudent;
11 |     mapStudent[1] = "student_one";
12 |     mapStudent[2] = "student_two";
13 |     mapStudent[3] = "student_three";
14 |     map<int, string>::iterator iter;
15 |     for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
16 |         cout<<iter->first<<' '<<iter->second<<endl;
17 | }
```

以上三种用法，虽然都可以实现数据的插入，但是它们是有区别的，当然了第一种和第二种在效果上是完成一样的，用insert函数插入数据，在数据的插入上涉及到集合的唯一性这个概念，即当map中有这个关键字时，insert操作是插入数据不了的，但是用数组方式就不同了，它可以覆盖以前该关键字对应的值，用程序说明

```
mapStudent.insert(map<int, string>::value_type (1, "student_one"));
```

```
mapStudent.insert(map<int, string>::value_type (1, "student_two"));
```

上面这两条语句执行后，map中1这个关键字对应的值是"student_one"，第二条语句并没有生效，那么这就涉及到我们怎么知道insert语句是否插入成功的问题了，可以用pair来获得是否插入成功，程序如下

```
pair<map<int, string>::iterator, bool> Insert_Pair;
```

```
Insert_Pair = mapStudent.insert(map<int, string>::value_type (1, "student_one"));
```

我们通过pair的第二个变量来知道是否插入成功，它的第一个变量返回的是一个map的迭代器，如果插入成功的话Insert_Pair.second应该是true的，否则为false。

下面给出完成代码，演示插入成功与否问题

```
1 | // 验证插入函数的作用效果
2 | #include <map>
3 | #include <string>
4 | #include <iostream>
5 | using namespace std;
6 |
7 | int main()
8 | {
9 |     map<int, string> mapStudent;
10 |     pair<map<int, string>::iterator, bool> Insert_Pair;
11 |     Insert_Pair = mapStudent.insert(pair<int, string>(1, "student_one"));
12 |     if(Insert_Pair.second == true)
13 |         cout<<"Insert Successfully"<<endl;
14 |     else
15 |         cout<<"Insert Failure"<<endl;
16 |     Insert_Pair = mapStudent.insert(pair<int, string>(1, "student_two"));
17 |     if(Insert_Pair.second == true)
18 |         cout<<"Insert Successfully"<<endl;
19 |     else
20 |         cout<<"Insert Failure"<<endl;
21 |     map<int, string>::iterator iter;
22 |     for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
23 |         cout<<iter->first<<' '<<iter->second<<endl;
24 | }
```

大家可以用如下程序，看下用数组插入在数据覆盖上的效果

```
1 | // 验证数组形式插入数据的效果
2 | #include <map>
3 | #include <string>
4 | #include <iostream>
5 | using namespace std;
6 |
7 | int main()
8 | {
9 |     map<int, string> mapStudent;
10 |     mapStudent[1] = "student_one";
11 |     mapStudent[1] = "student_two";
12 |     mapStudent[2] = "student_three";
13 |     map<int, string>::iterator iter;
14 |     for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
15 |         cout<<iter->first<<' '<<iter->second<<endl;
16 | }
```

3.4.5 查找、删除、交换

查找

```
1 | // 关键字查询，找到则返回指向该关键字的迭代器，否则返回指向end的迭代器
2 | // 根据map的类型，返回的迭代器为 iterator 或者 const_iterator
3 | iterator find (const key_type& k);
4 | const_iterator find (const key_type& k) const;
```

删除

```
1 | // 删除迭代器指向位置的键值对，并返回一个指向下一元素的迭代器
2 | iterator erase( iterator pos )
3 |
4 | // 删除一定范围内的元素，并返回一个指向下一元素的迭代器
5 | iterator erase( const_iterator first, const_iterator last );
6 |
7 | // 根据Key来进行删除， 返回删除的元素数量，在map里结果非0即1
8 | size_t erase( const key_type& key );
9 |
10 | // 清空map，清空后的size为0
11 | void clear();
```

交换

```
1 | // 就是两个map的内容互换
2 | void swap( map& other );
```

3.4.6 容量

```
1 | // 查询map是否为空
2 | bool empty();
3 |
4 | // 查询map中键值对的数量
5 | size_t size();
6 |
7 | // 查询map所能包含的最大键值对数量，和系统和应用库有关。
8 | // 此外，这并不意味着用户一定可以存这么多，很可能还没达到就已经开辟内存失败了
9 | size_t max_size();
10 |
```



```
11 | // 查询关键字为key的元素的个数，在map里结果非0即1
12 | size_t count( const Key& key ) const; //
```

3.4.7 排序

map中的元素是自动按Key升序排序，所以不能对map用sort函数；

这里要讲的是一点比较高深的用法了,排序问题，STL中默认是采用小于号来排序的，以上代码在排序上是不存在任何问题的，因为上面的关键字是int型，它本身支持小于号运算，在一些特殊情况，比如关键字是一个结构体或者自定义类，涉及到排序就会出现问 题，因为它没有小于号操作，insert等函数在编译的时候过 不去，下面给出两个方法解决这个问题。

3.4.7.1 小于号 < 重载

```
1 | #include <iostream>
2 | #include <string>
3 | #include <map>
4 | using namespace std;
5 |
6 | typedef struct tagStudentinfo
7 | {
8 |     int      niD;
9 |     string   strName;
10 |    bool operator < (tagStudentinfo const& _A) const
11 |    {
12 |        //这个函数指定排序策略，按niD排序，如果niD相等的话，按strName排序
13 |        if (niD < _A.niD) return true;
14 |        if (niD == _A.niD)
15 |            return strName.compare(_A.strName) < 0;
16 |        return false;
17 |    }
18 | }Studentinfo, *PStudentinfo; // 学生信息
19 |
20 | int main()
21 | {
22 |     int nSize;    //用学生信息映射分数
23 |     map<Studentinfo, int>mapStudent;
24 |     map<Studentinfo, int>::iterator iter;
25 |     Studentinfo studentinfo;
26 |     studentinfo.niD = 1;
27 |     studentinfo.strName = "student_one";
28 |     mapStudent.insert(pair<Studentinfo, int>(studentinfo, 90));
29 |     studentinfo.niD = 2;
30 |     studentinfo.strName = "student_two";
31 |     mapStudent.insert(pair<Studentinfo, int>(studentinfo, 80));
32 |     for (iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
33 |         cout << iter->first.niD << ' ' << iter->first.strName << ' ' << iter->second << endl;
34 |     return 0;
35 | }
```

3.4.7.2 仿函数的应用，这个时候结构体中没有直接的小于号重载

```
1 | // 第二种：仿函数的应用，这个时候结构体中没有直接的小于号重载，程序说明
2 |
3 | #include <iostream>
4 | #include <map>
5 | #include <string>
6 | using namespace std;
7 |
8 | typedef struct tagStudentinfo
9 | {
10 |     int      niD;
11 |     string   strName;
12 | }Studentinfo, *PStudentinfo; // 学生信息
13 |
14 | class sort
15 | {
16 | public:
17 |     bool operator() (Studentinfo const &_A, Studentinfo const &_B) const
18 |     {
19 |         if (_A.niD < _B.niD)
20 |             return true;
21 |         if (_A.niD == _B.niD)
22 |             return _A.strName.compare(_B.strName) < 0;
23 |         return false;
24 |     }
25 | };
26 |
27 | int main()
28 | {
29 |     //用学生信息映射分数
30 |     map<Studentinfo, int, sort>mapStudent;
31 |     map<Studentinfo, int>::iterator iter;
32 |     Studentinfo studentinfo;
33 |     studentinfo.niD = 1;
34 |     studentinfo.strName = "student_one";
35 |     mapStudent.insert(pair<Studentinfo, int>(studentinfo, 90));
36 |     studentinfo.niD = 2;
37 |     studentinfo.strName = "student_two";
38 |     mapStudent.insert(pair<Studentinfo, int>(studentinfo, 80));
39 |     for (iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
40 |         cout << iter->first.niD << ' ' << iter->first.strName << ' ' << iter->second << endl;
41 |     system("pause");
42 | }
```

3.4.8 unordered_map

在c++11标准前，c++标准库中只有一种map，但是它的底层实现并不是适合所有的场景，如果我们需要其他适合的map实现就不得不使用比如boost库等三方的实现，在c++11中加了一种map unordered_map,unordered_set,他们的实现有什么不同呢？

map底层采用的是红黑树的实现查询的时间复杂度为O(logn),看起来并没有unordered_map快，但是也要看实际的数据量，虽然unordered_map的查询从算法上分析比map快，但是它有一些其它的消耗，比如哈希函数的构造和分析，还有如果出现哈希冲突解决哈希冲突等等都有一定的消耗，因此unordered_map的效率在很大的程度上由它的hash函数算法决定，而红黑树的效率是一个稳定值。

unordered_map的底层采用哈希表的实现，查询的时间复杂度为是O(1)。所以unordered_map内部就是无序的，数据是按散列函数插入到槽里面去的，数据之间无顺序可言，如果我们不需要内部有序，这种实现是没有问题的。unordered_map属于关联式容器，采用std::pair保存key-value形式的数据。用法与map一致。特别的是，STL中的map因为是有序的 二叉树存储，所以对key值需要有大小的判断，当使用内置类型时，无需重载operator<；但是用用户自定义类型的话，就需要重载operator<。unordered_map全程使用不需要比较元素的key值的大小，但是，对于元素的==要有判断，又因为需要使用hash映射，所以，对于非内部类型，需要程序员为其定义这二者的内容，对于内部类型，就不需要了。unordered库使用“桶”来存储元素，散列值相同的被存储在一个桶里。当散列容器中有大量数据时，同一个桶里的数据也会增多，造成访问冲突，降低性能。为了提高散列容器的性能，unordered库会在插入元素是自动增加桶的数量，不需要用户指定。但是，用户也可以在构造函数或者rehash()函数中，指定最小的桶的数量。

还有另外一点从占用内存上来说因为unordered_map才用hash结构会有一定的内存损失，它的内存占用回高于map。

最后就是她们的场景了，首先如果你需要对map中的数据排序，就首选map，他会把你的数据按照key的自然排序排序（由于它的底层实现红黑树机制所以会排序），如果不需要排序，就看你对内存和cpu的选择了，不过一般都会选择unordered_map，它的查找效率会更高。

至于使用方法和函数，两者差不多，由于篇幅限制这里不再赘述，unordered_multimap用法亦可类推。

3.5 set/multiset

std::set 是关联容器，含有 Key 类型对象的已排序集。用比较函数compare进行排序。**搜索、移除和插入拥有对数复杂度**。set 通常以红黑树实现。

set容器内的元素会被自动排序，set与map不同，set中的元素即是键值又是实值，set不允许两个元素有相同的键值。不能通过set的迭代器去修改set元素，原因是修改元素会破坏se组织。当对容器中的元素进行插入或者删除时，操作之前的所有迭代器在操作之后依然有效。

由于set元素是排好序的，且默认为升序，因此当set集合中的元素为结构体或自定义类时，该结构体或自定义类必须实现运算符'<'的重载。

multiset特性及用法和set完全相同，唯一的差别在于它允许键值重复。

set和multiset的底层实现是一种高效的平衡二叉树，即红黑树（Red-Black Tree）。

3.5.1 set常用成员函数

- 1. begin()~返回指向第一个元素的迭代器
- 2. clear()~清除所有元素
- 3. count()~返回某个值元素的个数
- 4. empty()~如果集合为空，返回true
- 5. end()~返回指向最后一个元素的迭代器
- 6. equal_range()~返回集合中与给定值相等的上下限的两个迭代器
- 7. erase()~删除集合中的元素
- 8. find()~返回一个指向被查找到元素的迭代器
- 9. get_allocator()~返回集合的分配器
- 10. insert()~在集合中插入元素
- 11. lower_bound()~返回指向大于（或等于）某值的第一个元素的迭代器
- 12. key_comp()~返回一个用于元素间值比较的函数
- 13. max_size()~返回集合能容纳的元素的最大限值
- 14. rbegin()~返回指向集合中最后一个元素的反向迭代器
- 15. rend()~返回指向集合中第一个元素的反向迭代器
- 16. size()~集合中元素的数目
- 17. swap()~交换两个集合变量
- 18. upper_bound()~返回大于某个值元素的迭代器
- 19. value_comp()~返回一个用于比较元素间的值的函数

3.5.2 代码示例

- 以下代码涉及的内容:

- 1、set容器中，元素类型为基本类型，如何让set按照用户意愿来排序？
- 2、set容器中，如何让元素类型为自定义类型？
- 3、set容器的insert函数的返回值为什么类型？

```
1 | #include <iostream>
2 | #include <string>
3 | #include <set>
4 | using namespace std;
5 |
6 | /* 仿函数CompareSet, 在test02使用 */
7 | class CompareSet
8 | {
9 | public:
10 |     // 从大到小排序
11 |     bool operator()(int v1, int v2)
12 |     {
13 |         return v1 > v2;
14 |     }
15 |     // 从小到大排序
16 |     //bool operator()(int v1, int v2)
17 |     //{
18 |     //    return v1 < v2;
19 |     //}
20 | };
21 |
22 | /* Person类, 用于test03 */
23 | class Person
24 | {
25 |     friend ostream &operator<<(ostream &out, const Person &person);
26 | public:
27 |     Person(string name, int age)
28 |     {
29 |         mName = name;
30 |         mAge = age;
31 |     }
32 | public:
33 |     string mName;
34 |     int mAge;
35 | };
36 |
37 | ostream &operator<<(ostream &out, const Person &person)
38 | {
39 |     out << "name:" << person.mName << " age:" << person.mAge << endl;
40 |     return out;
41 | }
42 |
43 | /* 仿函数ComparePerson,用于test03 */
44 | class ComparePerson
45 | {
46 | public:
47 |     // 名字大的在前面, 如果名字相同, 年龄大的排前面
48 |     bool operator()(const Person &p1, const Person &p2)
49 |     {
50 |         if (p1.mName == p2.mName)
51 |         {
52 |             return p1.mAge > p2.mAge;
53 |         }
54 |         return p1.mName > p2.mName;
55 |     }
56 | };
57 |
58 | /* 打印set类型的函数模板 */
59 | template<typename T>
60 | void PrintSet(T &s)
61 | {
62 |     for (T::iterator iter = s.begin(); iter != s.end(); ++iter)
63 |         cout << *iter << " ";
64 |     cout << endl;
65 | }
66 |
67 | void test01()
68 | {
69 |     //set 容器默认从小到大排序
70 |     set<int> s;
71 |     s.insert(10);
72 |     s.insert(20);
73 |     s.insert(30);
74 |
75 |     // 输出set
76 |     PrintSet(s);
77 |     // 结果为:10 20 30
78 |
79 |     /* set的insert函数返回值为一个对组(pair)。
80 |      对组的第一个值first为set类型的迭代器;
81 |      1、若插入成功, 迭代器指向该元素。
82 |      2、若插入失败, 迭代器指向之前已经存在的元素
83 |
84 |      对组的第二个值second为bool类型;
85 |      1、若插入成功, bool值为true
86 |      2、若插入失败, bool值为false
87 |      */
88 |     pair<set<int>::iterator, bool> ret = s.insert(40);
89 |     if (true == ret.second)
90 |         cout << *ret.first << " 插入成功" << endl;
91 |     else
92 |         cout << *ret.first << " 插入失败" << endl;
```



```
93 | }
94 |
95 | void test02()
96 | {
97 |     /* 如果想让set容器从小到大排序, 需要给set容
98 |        器提供一个仿函数, 本例的仿函数为CompareSet
99 |        */
100 |    set<int, CompareSet> s;
101 |    s.insert(10);
102 |    s.insert(20);
103 |    s.insert(30);
104 |
105 |    //打印Set
106 |    PrintSet(s);
107 |    //结果为:30,20,10
108 | }
109 |
110 | void test03()
111 | {
112 |     /* set元素类型为Person, 当set元素类型为自定义类型的时候
113 |        必须给set提供一个仿函数, 用于比较自定义类型的大小,
114 |        否则无法通过编译
115 |        */
116 |    set<Person, ComparePerson> s;
117 |    s.insert(Person("John", 22));
118 |    s.insert(Person("Peter", 25));
119 |    s.insert(Person("Marry", 18));
120 |    s.insert(Person("Peter", 36));
121 |
122 |    //打印Set
123 |    PrintSet(s);
124 | }
125 |
126 | int main(void)
127 | {
128 |     //test01();
129 |     //test02();
130 |     //test03();
131 |     return 0;
132 | }
```

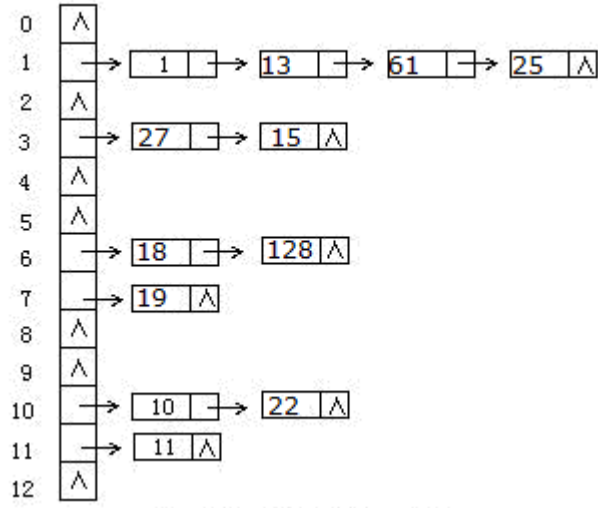
- multiset容器的insert函数返回值为什么？

```
1 | #include <iostream>
2 | #include <set>
3 | using namespace std;
4 |
5 | /* 打印set类型的函数模板 */
6 | template<typename T>
7 | void PrintSet(T &s)
8 | {
9 |     for (T::iterator iter = s.begin(); iter != s.end(); ++iter)
10 |         cout << *iter << " ";
11 |     cout << endl;
12 | }
13 |
14 | void test(void)
15 | {
16 |     multiset<int> s;
17 |     s.insert(10);
18 |     s.insert(20);
19 |     s.insert(30);
20 |
21 |     //打印multiset
22 |     PrintSet(s);
23 |
24 |     /* multiset的insert函数返回值为multiset类型的迭代器,
25 |        指向新插入的元素。multiset允许插入相同的值, 因此
26 |        插入一定成功, 因此不需要返回bool类型。
27 |        */
28 |     multiset<int>::iterator iter = s.insert(10);
29 |
30 |     cout << *iter << endl;
31 | }
32 |
33 | int main(void)
34 | {
35 |     test();
36 |     return 0;
37 | }
```

3.5.3 unordered_set

C++ 11中出现了两种新的关联容器:unordered_set和unordered_map, 其内部实现与set和map大有不同, set和map内部实现是基于RB-Tree, 而unordered_set和unordered_map内部实现是基于哈希表(hashtable), 由于unordered_set和unordered_map内部实现的公共接口大致相同, 所以本文以unordered_set为例。

unordered_set是基于哈希表, 因此要了解unordered_set, 就必须了解哈希表的机制。哈希表是根据关键码值而进行直接访问的数据结构, 通过相应的哈希函数(也称散列函数)处理关键字得到相应的关键码值, 关键码值对应着一个特定位置, 用该位置来存取相应的信息, 这样就能以较快的速度获取关键字的信息。比如: 现有公司员工的个人信息(包括年龄), 需要查询某个年龄的员工个数。由于人的年龄范围大约在[0, 200], 所以可以开一个200大小的数组, 然后通过哈希函数 $f(key) = key$ 得到key对应的key-value, 这样就能完成统计某个年龄的员工个数。而在这个例子中, 也存在这样一个问题, 两个员工的年龄相同, 但其他信息(如: 名字、身份证)不同, 通过前面说的哈希函数, 会发现其都位于数组的相同位置, 这里, 就涉及到“冲突”。准确来说, 冲突是不可避免的, 而解决冲突的方法常见的有: 开发地址法、再散列法、链地址法(也称拉链法)。而unordered_set内部解决冲突采用的是——链地址法, 当用冲突发生时把具有同一关键码的数据组成一个链表。下图展示了链地址法的使用:



使用unordered_set需要包含#include<unordered_set>头文件, 同unordered_map类似, 用法没有什么太大的区别, 参考set/multiset。

除此之外unordered_multiset也是一种可选的容器。

reference:

<http://www.runoob.com/w3cnote/cpp-vector-container-analysis.html>

<https://blog.csdn.net/tianshuai1111/article/details/7687983>