c++中容器总结 - 我的学习笔记

原创

2013-08-18 15:26:26

C++中的容器大致可以分为两个大类：顺序容器和关联容器。顺序容器中有包含有顺序容器适配器。

顺序容器：将单一类型元素聚集起来成为容器，然后根据位置来存储和访问这些元素。主要有vector、list、deque（双端队列）。顺序容器适配器：stack、queue和priority_queue。

关联容器：支持通过键来高效地查找和读取元素。主要有：pair、set、map、multiset和multimap。

接下来依次对于各种容器做详细的介绍。

一、顺序容器1、顺序容器定义
为了定义一个容器类型的对象，必须先包含相关的头文件：

定义vector：#include <vector>

定义list：#include <list>

定义deque：#include <deque>

定义示例

2、顺序容器初始化

函数模板	意义
C<T> c;	创建一个名为c的空容器。C是容器类型名，如vector，T是元素类型，如int或string适用于所有容器。
C c(c2);	创建容器c2的副本c；c和c2必须具有相同的容器类型，并存放相同类型的元素。适用于所有容器。
C c(b,e);	创建c，其元素是迭代器b和e标示的范围内元素的副本。适用于所有容器。
C c(n,t);	用n个值为t的元素创建容器c，其中值t必须是容器类型C的元素类型的值，或者是可转换为该类型的值。

	只适用于顺序容器
C c(n);	创建有 n 个值初始化元素的容器 c。 只适用于顺序容器

初始化示例：

```
//初始化为一个容器的副本
vector<int> vi;
vector<int> vi2(vi); //利用vi来初始化vi2
//初始化为一段元素的副本
char*words[] = {"stately", "plump", "buck", "mulligan"};
size_twords_size = sizeof(words)/sizeof(char*);
list<string> words2(words, words + words_size);//分配和初始化指定数目的元素
constlist<int>::size_type list_size = 64;
list<string> slist(list_size, "a"); // 64 strings, each is a
```

3、顺序容器支持的指针运算

①所有顺序都支持的指针运行

表达式	意义
*iter	返回迭代器 iter 所指向的元素的引用
iter->mem	对iter进行解引用，获取指定元素中名为mem的成员。等效于(*iter).mem
++iter iter++	给 iter 加 1，使其指向容器里的下一个元素
--iter iter--	给 iter 减 1，使其指向容器里的前一个元素
iter==iter2 iter1!=iter2	比较两个迭代器是否相等（或不等）。当两个迭代器指向同一个 容器中的同一个元素，或者当它们都指向同一个容器的超出末端 的下一位置时，两个迭代器相等

②vector 和 deque 容器的迭代器提供额外的运算

表达式	意义
iter + n iter - n	在迭代器上加（减）整数值 n，将产生指向容器中前面（后面）第 n 个元素的迭代器。新计算出来的迭代器必须指向容器中的元素或超出容器末端的下一位置
iter1+=iter2 iter1-=iter2	这里迭代器加减法的复合赋值运算：将 iter1 加上或减去 iter2 的运算结果赋给 iter1
iter1 - iter2	两个迭代器的减法，其运算结果加上右边的迭代器即得左边的迭代器。这两个迭代器必须指向同一个容器中的元素或超出容器末端的下一位置
>, >=, <, <=	迭代器的关系操作符。当一个迭代器指向的元素在容器中位于另一个迭代器指向的元素之前，则前一个迭代器小于后一个迭代器。关系操作符的两个迭代器必须指向同一个容器中的元素或超出容器末端的下一位置

③迭代器失效：一些容器操作会修改容器的内在状态或移动容器内的元素。这样的操作使所有指向被移动的元素的迭代器失效，也可能同时使其他迭代器失效。使用无效迭代器是没有定义的，可能会导致与悬垂指针相同的问题。

④begin和end成员：begin和end操作产生指向容器内第一个元素和最后一个元素的下一位置的迭代器。

函数名	意义
c.begin()	返回一个迭代器，它指向容器 c 的第一个元素
c.end()	返回一个迭代器，它指向容器 c 的最后一个元素的下一位置
c.rbegin()	返回一个逆序迭代器，它指向容器 c 的最后一个元素
c.rend()	返回一个逆序迭代器，它指向容器 c 的第一个元素前面的位置

3、顺序容器操作

①添加元素

函数名	意义
c.push_back(t)	在容器c的尾部添加值为t的元素。返回void 类型
c.push_front(t)	在容器c的前端添加值为t的元素。返回void 类型 只适用于list和deque容器类型。
c.insert(p,t)	在迭代器p所指向的元素前面插入值为t的新元素。返回指向新添加元素的迭代器。
c.insert(p,n,t)	在迭代器p所指向的元素前面插入n个值为t的新元素。返回void 类型
c.insert(p,b,e)	在迭代器p所指向的元素前面插入由迭代器b和e标记的范围内的元素。返回 void 类型

添加元素示例：

```
//在容器首部或者尾部添加数据
list<int> ilist;
ilist.push_back(ix);//尾部添加
ilist.push_front(ix);//首部添加
//在容器中指定位置添加元素list<string> lst;list<string>::iterator iter = lst.begin();
while (cin >> word)
iter = lst.insert(iter, word); // 和push_front意义一样
//插入一段元素list<string> slist;
string sarray[4] = {"quasi", "simba", "frollo", "scar"};
slist.insert(slist.end(), 10, "A");//尾部前添加十个元素都是A
list<string>::iterator slist_iter = slist.begin();
slist.insert(slist_iter, sarray+2, sarray+4);//指针范围添加
```

②容器大小的操作

函数名	意义
c.size()	返回容器c中元素个数。返回类型为 c::size_type
c.max_size()	返回容器c可容纳的最多元素个数，返回类型为c::size_type
c.empty()	返回标记容器大小是否为0的布尔值

c.resize(n)	调整容器c的长度大小，使其能容纳n个元素，如果n<c.size()，则删除多出来的元素；否则，添加采用值初始化的新元素
c.resize(n,t)	调整容器c的长度大小，使其能容纳n个元素。所有新添加的元素值都为t

示例：

```
list<int> ilist(10, 1);
ilist.resize(15); // 尾部添加五个元素，值都为0
ilist.resize(25, -1); // 再在尾部添加十个元素，元素为-1
ilist.resize(5); // 从尾部删除20个元素
```

③访问元素

函数名	意义
c.back()	返回容器 c 的最后一个元素的引用。如果 c 为空，则该操作未定义
c.front()	返回容器 c 的第一个元素的引用。如果 c 为空，则该操作未定义
c[n]	返回下标为 n 的元素的引用。如果 n <0 或 n >= c.size()，则该操作未定义 只适用于 vector 和 deque 容器
c.at(n)	返回下标为 n 的元素的引用。如果下标越界，则该操作未定义 只适用于 vector 和 deque 容器

```
//删除第一个或最后一个元素
list<int> li;
for(int i=0;i<10;i++)list.push_back(i);
li.pop_front();//删除第一个元素
li.pop_back(); //删除最后一个元素
//删除容器内的一个元素
list<int>::iterator iter =li.begin();
if(iter!= li.end())li.erase(iter);
//删除容器内所有元素li.clear();
```

⑤赋值与swap

函数名	意义

c1 = c2	删除容器c1的所有元素，然后将c2的元素复制给c1。c1和c2的类型（包括容器类型和元素类型）必须相同
c1.swap(c2)	交换内容：调用完该函数后，c1中存放的是 c2 原来的元素，c2中存放的则是c1原来的元素。c1和c2的类型必须相同。该函数的执行速度通常要比将c2复制到c1的操作快
c.assign(b,e)	重新设置c的元素：将迭代器b和e标记的范围内所有的元素复制到c中。b和e必须不是指向c中元素的迭代器
c.assign(n,t)	将容器c重新设置为存储n个值为t的元素

赋值assign示例：

```
list<string> sl1,sl2;
for(int i=0;i<10;i++)sl2.push_back("a");
sl1.assign(sl2.begin(),sl2.end());//用sl2的指针范围赋值，sl1中十个元素都为a
sl1.assign(10, "A"); //s1被重新赋值，拥有十个元素，都为A
```

swap示例：

```
vector<string> vs1(3); // vs1有3个元素
vector<string> vs(5); // vs2有5个元素
vs1.swap(vs2);//执行后,vs1中5个元素，而vs2则存3个元素。
```

⑥vector的自增长：capacity 和 reserve 成员

为了提高vector的效率，不用每次添加元素都重新分配空间。vector会在分配空间时候预分配大于需要的空间。vector 类提供了两个成员函数：capacity 和reserve 使程序员可与 vector 容器内存分配的实现部分交互工作。

capacity操作：获取在容器需要分配更多的存储空间之前能够存储的元素总数

reserve操作：告诉vector容器应该预留多少个元素的存储空间

capacity（容量）与size（长度）的区别：size指容器当前拥有的元素个数，而capacity则指容器在必须分配新存储空间之前可以存储的元素总数。capacity是比size大的一般情况下。

示例：

```
vector<int> ivec;
cout << "ivec: size: " << ivec.size()
<< " capacity: " << ivec.capacity() << endl;//都为0
for (vector<int>::size_type ix = 0; ix != 24; ++ix)ivec.push_back(ix);
cout << "ivec: size: " << ivec.size()
<< " capacity: " << ivec.capacity() << endl;//capacity大于size
```

可以通过函数reserve()来操作预留空间

```
1
2
//在之前一段代码的基础上
ivec.reserve（ivec.capacity()+50）;//为ivec增加了50的预留空间
```

另外：如果不手动操作来预留空间，每当 vector 容器不得不分配新的存储空间时，以加倍当前容量的分配策略实现重新分配。

4、容器的选用

选择容器类型的常规法则：

①如果程序要求随机访问元素，则应使用 vector 或 deque 容器。

②如果程序必须在容器的中间位置插入或删除元素，则应采用 list 容器。

③如果程序不是在容器的中间位置，而是在容器首部或尾部插入或删除元素，则应采用 deque 容器。

④如果只需在读取输入时在容器的中间位置插入元素，然后需要随机访问元素，则可考虑在输入时将元素读入到一个 list 容器，接着对此容器重新排序，使其适合顺序访问，然后将排序后的 list 容器复制到一个 vector容器。

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，选择何种容器取决于下面两种操作付出的相对代价：随机访问 list 容器元素的代价，以及在 vector 或 deque 容器中插入 / 删除元素时复制元素的代价。通常来说，应用中占优势的操作（程序中更多使用的是访问操作还是插入 / 删除操作）将决定应该什么类型的容器。

5、容器适配器

①适配器通用的操作和类型

名称	意义
size_type	一种类型，足以存储此适配器类型最大对象的长度
value_type	元素类型
container_type	基础容器的类型，适配器在此容器类型上实现
A a;	创建一个新空适配器，命名为 a
A a(c);	创建一个名为 a 的新适配器，初始化为容器 c 的副本
关系操作符	所有适配器都支持全部关系操作符：==、 !=、 <、 <=、 >、>=

②适配器的初始化

所有适配器都定义了两个构造函数：默认构造函数用于创建空对象，而带一个容器参数的构造函数将参数容器的副本作为其基础值。

默认的stack和queue都基于deque容器实现，而priority_queue则在vector容器上实现。
示例：

```
vector<int> vi;
deque<int> deq;
stack<int> stk(deq); //用deq初始化stk
stack<int> stk1(vi); //报错
```

③适配器的操作

栈适配器：

函数名	意义
s.empty()	如果栈为空，则返回true，否则返回stack
s.size()	返回栈中元素的个数
s.pop()	删除栈顶元素的值，但不返回其值
s.top()	返回栈顶元素的值，但不删除该元素
s.push(item)	在栈顶压入新元素

队列和优先级队列：

函数名	意义
q.empty()	如果队列为空，则返回true，否则返回false
q.size()	返回队列中元素的个数
q.pop()	删除队首元素，但不返回其值
q.front()	返回队首元素的值，但不删除该元素 该操作只适用于队列
q.back()	返回队尾元素的值，但不删除该元素 该操作只适用于队列

q.top()	返回具有最高优先级的元素值，但不删除该元素 该操作只适用于优先级队列
q.push(item)	对于queue，在队尾压入一个新元素，对于priority_quue，在基于优先级的适当位置插入新元素

二、关联容器1、pair

①pairs类型提供的操作

操作名	意义
pair<T1, T2>p1	创建一个空的pair对象，它的两个元素分别是T1和T2类型，采用值初始化
pair<T1, T2> p1(v1, v2)	创建一个pair对象，它的两个元素分别是T1和T2，其中first成员初始化为v1，而second成员初始化为v2
make_pair(v1,v2)	以v1和v2值创建一个新pair对象，其元素类型分别是v1和v2的类型
p1 < p2	两个pair对象之间的小于运算，其定义遵循字典次序：如果 p1.first< p2.first或者!(p2.first<p1.first)&&p1.second<p2.second，则返回true
p1 == p2	如果两个pair对象的first和second成员依次相等，则这两个对象相等。该运算使用其元素的==操作符
p.first	返回p中名为first的（公有）数据成员
p.second	返回p的名为second的（公有）数据成员

②pairs类型定义和初始化

1

```
pair<string, string> test("A", "B");
```

③pairs其他操作

```
//pairs对象的操作string firstBook;
if (author.first == "James" && author.second == "Joyce")firstBook = "Stephen Hero";
//生成新的pair对象pair<string, string> next_auth;
next_auth = make_pair("A","B");//第一种方法
```

```
next_auth = pair<string, string>("A","B"); //第二种方法
cin>>next_auth.first>>next_auth.second;//第三种方法
```

2、map

map 是键－值对的集合。map 类型通常可理解为关联数组：可使用键作为下标来获取一个值，正如内置数组类型一样。而关联的本质在于元素的值与某个特定的键相关联，而并非通过元素在数组中的位置来获取。

①map对象的定义

1

```
map<string, int> word_count;
```

这个语句定义了一个名为 word_count 的 map 对象，由 string 类型的键索引，关联的值则int型。

map访问：对迭代器进行解引用时，将获得一个引用，指向容器中一个pair<const string,int>类

型的值。通过pair类型的访问方式进行访问。

②map的构造函数

函数名	意义
map<k, v>m	创建一个名为m的空map对象，其键和值的类型分别为k和v
map<k, v> m(m2)	创建m2的副本m，m与m2必须有相同的键类型和值类型
map<k, v> m(b, e)	创建map类型的对象m，存储迭代器b和e标记的范围内所有元素的副本。元素的类型必须能转换为pair<const k, v>

③键类型的约束

在使用关联容器时，它的键不但有一个类型，而且还有一个相关的比较函数。所用的比较函数必须在键类型上定义严格弱排序（strict weak ordering）：可理解为键类型数据上的“小于”关系，虽然实际上可以选择将比较函数设计得更复杂。

对于键类型，唯一的约束就是必须支持 < 操作符，至于是否支持其他的关系或相等运算，则不作要求。

④map类定义的类型

类型	意义

map<K,V>::key_type	在 map 容器中，用做索引的键的类型
map<K,V>::mapped_type	在 map 容器中，键所关联的值的类型
map<K,V>::value_type	一个pair类型，它的first元素具有const map<K,V>::key_type类型，而second元素则为map<K,V>::mapped_type 类型

value_type是存储元素的键以及值的pair类型，而且键为const。例如，word_count 数组的value_type为pair<const string,int> 类型。value_type 是 pair 类型，它的值成员可以修改，但键成员不能修改。

⑤map添加元素

添加元素有两种方法：1、先用下标操作符获取元素，然后给获取的元素赋值 2、使用insert成员函数实现

下标操作添加元素：如果该键已在容器中，则 map 的下标运算与 vector 的下标运算行为相同：返回该键所关联的值。只有在所查找的键不存在时，map 容器才为该键创建一个新的元素，并将它插入到此 map 对象中。此时，所关联的值采用值初始化：类类型的元素用默认构造函数初始化，而内置类型的元素初始化为 0。

insert 操作：

函数名	意义
m.insert(e)	e是一个用在m上的value_type 类型的值。如果键（e.first不在m中，则插入一个值为e.second 的新元素；如果该键在m中已存在，则保持m不变。该函数返回一个pair类型对象，包含指向键为e.first的元素的map迭代器，以及一个 bool 类型的对象，表示是否插入了该元素
m.insert (beg,end)	beg和end是标记元素范围的迭代器，其中的元素必须为m.value_type 类型的键－值对。对于该范围内的所有元素，如果它的键在 m 中不存在，则将该键及其关联的值插入到 m。返回 void 类型
m.insert (iter,e)	e是一个用在m上的 value_type 类型的值。如果键（e.first）不在m中，则创建新元素，并以迭代器iter为起点搜索新元素存储的位置。返回一个迭代器，指向m中具有给定键的元素

示例：

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

```
word_count.insert(make_pair("Anna", 1));
```

insert的返回值：包含一个迭代器和一个bool值的pair对象，其中迭代器指向map中具有相应键的元素，而bool值则表示是否插入了该元素。如果该键已在容器中，则其关联的值保持不变，返回的bool值为true。在这两种情况下，迭代器都将指向具有给定键的元素。

1
2

```
pair<map<string, int>::iterator, bool> ret =
word_count.insert(make_pair(word, 1));
```

ret存储insert函数返回的pair对象。该pair的first成员是一个map迭代器，指向插入的键。ret.first从insert返回的pair对象中获取 map 迭代器；ret.second从insert返回是否插入了该元素。

⑥查找并读取map中的元素

map中使用下标存在一个很危险的副作用：如果该键不在 map 容器中，那么下标操作会插入一个具有该键的新元素。所以map 容器提供了两个操作：count 和 find，用于检查某个键是否存在而不会插入该键。

函数名	意义
m.count(k)	返回 m 中 k 的出现次数
m.find(k)	如果m容器中存在按k索引的元素，则返回指向该元素的迭代器。如果不存在，则返回超出末端迭代器。

1
2
3
4

```
int occurs = 0;
if (word_count.count("foobar"))occurs = word_count["foobar"];
map<string,int>::iterator it = word_count.find("foobar");
if (it != word_count.end())occurs = it->second;
```

⑦从map对象中删除元素

函数名	意义
m.erase(k)	删除m中键为k的元素。返回size_type类型的值，表示删除的元素个数
m.erase(p)	从m中删除迭代器p所指向的元素。p必须指向m中确实存在的元素，而且不能等于m.end()。返回void

m.erase(b, e)	从m中删除一段范围内的元素，该范围由迭代器对b和e标记。b和e必须标记m中的一段有效范围：即b和e都必须指向m中的元素或最后一个元素的下一个位置。而且，b和e要么相等（此时删除的范围为空），要么b所指向的元素必须出在e所 指向的元素之前。返回 void 类型

1
2
3
4

3、set

①set容器的定义和使用

set 容器的每个键都只能对应一个元素。以一段范围的元素初始化set对象，或在set对象中插入一组元素时，对于每个键，事实上都只添加了一个元素。

1
2
3
4
5
6
7
8

```
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
ivec.push_back(i);ivec.push_back(i);}
set<int> iset(ivec.begin(), ivec.end());
cout << ivec.size() << endl; //20个
cout << iset.size() << endl; // 10个
```

②在set中添加元素

1
2
3
4

```
set<string> set1;
set1.insert("the"); //第一种方法：直接添加
set<int> iset2;
iset2.insert(ivec.begin(), ivec.end());//第二中方法：通过指针迭代器
```

③从set中获取元素

set 容器不提供下标操作符。为了通过键从 set 中获取元素，可使用 find运算。如果只需简单地判断

某个元素是否存在，同样可以使用 count 运算，返回 set 中该键对应的元素个数。当然，对于 set 容器，count 的返回值只能是1（该元素存在）或 0（该元素不存在）

```
1
2
3
4
5
6
set<int> iset;
for(int i = 0; i<10; i++)iset.insert(i);
iset.find(1) // 返回指向元素内容为1的指针
iset.find(11) // 返回指针iset.end()
iset.count(1) // 存在，返回1
iset.count(11) // 不存在，返回0
```

3、multimap 和 multiset

关联容器 map 和 set 的元素是按顺序存储的。而 multimap 和multset 也一样。因此，在multimap和multiset容器中，如果某个键对应多个实例，则这些实例在容器中将相邻存放。迭代遍历multimap或multiset容器时，可保证依次返回特定键所关联的所有元素。

①迭代器的关联容器操作

函数名	意义
m.lower_bound(k)	返回一个迭代器，指向键不小于 k 的第一个元素
m.upper_bound(k)	返回一个迭代器，指向键大于 k 的第一个元素
m.equal_range(k)	返回一个迭代器的 pair 对象。它的 first 成员等价于 m.lower_bound(k)。而 second 成员则等价于 m.upper_bound(k)