C++容器学习

2021年7月28日 9:45

C++STL中容器的使用全面总结

1、容器的定义与介绍

在数据存储上，有一种对象类型，它可以持有其它对象或指向其它对象的指针，这种对象类型就叫做容器。很简单，容器就是保存其它对象的对 象，当然这是一个朴素的理解，这种“对象”还包含了一系列处理“其它对象”的方法，因为这些方法在程序的设计上会经常被用到，所以容器也体现了一个好处， 就是“容器类是一种对特定代码重用问题的良好的解决方案”。

容器还有另一个特点是容器可以自行扩展。在解决问题时我们常常不知道我们需要存储多少个对象，也就是说我们不知道应该创建多大的内存空间来保存我们的对象。 显然，数组在这一方面也力不从心。容器的优势就在这里，它不需要你预先告诉它你要存储多少对象，只要你创建一个容器对象，并合理的调用它所提供的方法，所 有的处理细节将由容器来自身完成。它可以为你申请内存或释放内存，并且用最优的算法来执行您的命令。

容器是随着面向对象语言的诞生而提出的，容器类在面向对象语言中特别重要，甚至它被认为是早期面向对象语言的基础。在现在几乎所有的面向对象的语言中也都伴随着一个容器集，在C++ 中，就是标准模板库（STL ）。

和其它语言不一样，C++ 中处理容器是采用基于模板的方式。标准C++ 库中的容器提供了多种数据结构，这些数据结构可以与标准算法一起很好的工作，这为我们的软件开发提供了良好的支持！

2、容器的分类

STL 对定义的通用容器分三类：顺序性容器、关联式容器和容器适配器。

顺序性容器是一种各元素之间有顺序关系的线性表，是一种线性结构的可序群集。顺序性容器中的每个元素均有固定的位置，除非用删除或插入的操作改变这个位置。这个位置和元素本身无关，而和操作的时间和地点有关，顺序性容器不会根据元素的特点排序而是直接保存了元素操作时的逻辑顺序。比如我们一次性对一个顺序性容器追加三个元素，这三个元素在容器中的相对位置和追加时的逻辑次序是一致的。

关联式容器和顺序性容器不一样，关联式容器是 非线性的树结构，更准确的说是二叉树结构。各元素之间没有严格的物理上的顺序关系，也就是说元素在容器中并没有保存元素置 入容器时的逻辑顺序。但是关联式容器提供了另一种根据元素特点排序的功能，这样迭代器就能根据元素的特点“顺序地”获取元素。

关联式容器另一个显著的特点是它是以键值的方式来保存数据，就是说它能把关键字和值关联起来保存，而顺序性容器只能保存一种（可以认为它只保存关键字，也可以认为它只保存值）。这在下面具体的容器类中可以说明这一点。

容器适配器是一个比较抽象的概念，C++的解释是：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。容器适配器是让一种已存在的容器类型采用另一种不同的抽象类型的工作方式来实现的一种机制。其实仅是发生了接口转换。那么你可以把它理解为容器的容器，它实质还是一个容器，只是他不依赖于具体的标准容器类型，可以理解是容器的模版。或者把它理解为容器的接口，而适配器具体采用哪种容器类型去实现，在定义适配器的时候可以由你决定。

标准容器类	特点
顺序性容器	
vector	从后面快速的插入与删除，直接访问任何元素。
deque	从前面或后面快速的插入与删除，直接访问任何元素。
list	双链表，从任何地方快速插入与删除。
关联容器	

set	快速查找，不允许重复值。
multiset	快速查找，允许重复值。
map	一对多映射，基于关键字快速查找，不允许重复值。
multimap	一对多映射，基于关键字快速查找，允许重复值。
容器适配器	
stack	后进先出。
queue	先进先出。
priority_queue	最高优先级元素总是第一个出列。

3、array（STL array）容器用法总结

3.1 array的简单介绍

array容器是C++11标准中新增的序列容器，简单地理解，它就是在C++普通数组的基础上，添加了一些成员函数和全局函数。在使用上，它比普通数组更安全，且效率并没有因此变差。

和其他容器不同，array容器的大小是固定的，无法动态地扩展或收缩，这也就意味着，在使用该容器的过程无法借由增加或移除元素而改变其大小，它只允许访问或替换存储的元素。

array容器以类模板的形式定义在<array>头文件，并位于命名空间std中。

```
namespace std{
    template <typename T, size_t N>
    class array;
}
```

3.2 array基本函数使用

3.2.1构造函数

array容器有多种初始化方式。

```
std::array<double, 10> values; //由此，就创建好了一个名为 values 的 array 容器，其包含 10 个浮点型元素。但是，由于未显式指定这 10 个元素的值，因此使用这种方式创建的容器中，各个元素的值是不确定的（array 容器不会做默认初始化操作）。
std::array<double, 10> values {}; //将所有的元素初始化为 0 或者和默认元素类型等效的值.
std::array<double, 10> values {0.5, 1.0, 1.5, 2.0}; //初始化了前 4 个元素，剩余的元素都会被初始化为 0.0
```

3.2.2 array中的函数

函数成员	**函数功能**
begin()	返回指向容器中第一个元素的随机访问迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的随机访问迭代器，通常和 begin() 结合使用。
rbegin()	返回指向最后一个元素的随机访问迭代器。
rend()	返回指向第一个元素所在位置前一个位置的随机访问迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
size()	返回容器中当前元素的数量，其值始终等于初始化array类的第二个模板参数N。
max_size()	返回容器可容纳元素的最大数量，其值始终等于初始化array类的第二个模板参数N。
empty()	判断容器是否为空，和通过size()==0的判断条件功能相同，但其效率可能更快。
at(n)	返回容器中n位置处元素的引用，该函数自动检查n是否在有效范围内，如果不是则抛出out_of_range异常。
front()	返回第一个元素的引用，该函数不适合于空的array容器。

back()	返回最后一个元素的引用，该函数同样不适合于空的array容器。
data()	返回一个指向容器首个元素的指针。利用该指针，可实现复制容器中所有元素等类似功能。
fill(val)	将val这个值赋值给容器中的每个元素。
array1.swap(array2)	交换array1和array2容器中的所有元素，但前提是它们具有相同的长度和类型。

```
#include <iostream>
//需要引入 array 头文件
#include <array>
using namespace std;
int main()
{
    std::array<int, 4> values{};
    //初始化 values 容器为 {0,1,2,3}
    for (int i = 0; i < values.size(); i++) {
        values.at(i) = i;
    }
    //使用 get() 重载函数输出指定位置元素
    cout << get<3>(values) << endl;
    //如果容器不为空，则输出容器中所有的元素
    if (!values.empty()) {
        for (auto val = values.begin(); val < values.end(); val++) {
            cout << *val << " ";
        }
    }
}
```

注意，代码中的auto关键字，可以使编译器自动判定变量的类型。运行这段代码，输出结果为：

```
3
0 1 2 3
```

3.2.3 STL array随机访问迭代器

（1）begin()/end()和cbegin()/cend()

```
#include <iostream>
//需要引入 array 头文件
#include <array>
using namespace std;
int main()
{
    array<int, 5>values;
    int h = 1;
    auto first = values.begin();
    auto last = values.end();
    //初始化 values 容器为{1,2,3,4,5}
    while (first != last)
    {
        *first = h;
        ++first;
        h++;
    }

    first = values.begin();
    while (first != last)
    {
        cout << *first << " ";
        ++first;
    }
    return 0;
}
```

可以看出，迭代器对象是由 array 对象的成员函数 begin() 和 end() 返回的。我们可以像使用普通指针那样使用迭代器对象。比如代码中，在保存了元素值后，使用前缀++运算符对first进行自增，当first等于end时，所有的元素都被设完值，循环结束。

与此同时，还可以使用全局的begin()和end()函数来从容器中获取迭代器，因为当操作对象为array容器时，它们和begin()/end()成员函数是通用的。所以上面代码中，first和last还可以像下面这样定义：

```
auto first = std::begin(values);
auto last = std::end (values);
```

除此之外，array模板类还提供了cbegin()和cend()成员函数，它们和begin()/end()唯一不同的是，前者返回的是const类型的正向迭代器，这就意味着，有cbegin()和cend()成员函数返回的迭代器，可以用来遍历容器内的元素，也可以访问元素，但是不能对所存储的元素进行修改。

```
#include <iostream>
//需要引入 array 头文件
#include <array>
using namespace std;
int main()
{
```

```
    array<int, 5>values{1,2,3,4,5};
    int h = 1;
    auto first = values.cbegin();
    auto last = values.cend();

    //由于 *first 为 const 类型，不能用来修改元素
    //*first = 10;

    //遍历容器并输出容器中所有元素
    while (first != last)
    {
        //可以使用 const 类型迭代器访问元素
        cout << *first << " ";
        ++first;
    }
    return 0;
}
```

此程序的第14行代码中，我们尝试使用first迭代器修改values容器中的值，如果取消注释并运行此程序，编译器会提示你“不能给常量赋值”，即*first是const类型常量，所以这么做是不对的。但17~22行代码遍历并访问容器的行为，是允许的。

（2）rbegin()/rend()和crbegin()/crend()

```
#include <iostream>
//需要引入 array 头文件
#include <array>
using namespace std;
int main()
{
    array<int, 5>values;
    int h = 1;
    auto first = values.rbegin();
    auto last = values.rend();
    //初始化 values 容器为 {5,4,3,2,1}
    while (first != last)
    {
        *first = h;
        ++first;
        h++;
    }
    //重新遍历容器，并输入各个元素
    first = values.rbegin();
    while (first != last)
    {
        cout << *first << " ";
        ++first;
    }
    return 0;
}
```

3.2.4 STL array容器访问元素的几种方式

（1）访问array容器中单个元素

可以通过容器名[]的方式直接访问和使用容器中的元素，这和C++标准数组访问元素的方式相同。

```
values[4] = values[3] + 2.0*values[1];
```

此行代码中，第5个元素的值被赋值为右边表达式的值。需要注意的是，使用如上这样方式，由于没有做任何边界检查，所以即便使用越界的索引值去访问或存储元素，也不会被检测到。

为了能够有效地避免越界访问的情况，可以使用array容器提供的at()成员函数。

```
values.at (4) = values.at(3) + 2.0*values.at(1);
```

这行代码和前一行语句实现的功能相同，其次当传给 at() 的索引是一个越界值时，程序会抛出 std::out_of_range 异常。因此当需要访问容器中某个指定元素时，建议大家使用 at()，除非确定索引没有越界。

除此之外，array容器还提供了get<n>模板函数，它是一个辅助函数，能够获取到容器的第n个元素。需要注意的是，该模板函数中，参数的实参必须是一个在编译时可以确定的常量表达式，所以它不能是一个循环变量。也就是说，它只能访问模板参数指定的元素，编译器在编译时会对它进行检查。

```
#include <iostream>
#include <array>
#include <string>
using namespace std;
int main()
{
    array<string, 5> words{ "one","two","three","four","five" };
    cout << get<3>(words) << endl; // Output words[3]
    //cout << get<6>(words) << std::endl; //越界，会发生编译错误
    return 0;
}
```

另外，array 容器提供了 data() 成员函数，通过调用该函数可以得到指向容器首个元素的指针。通过该指针，我们可以获得容器中的各个元素，例如：

```
#include <iostream>
#include <array>
using namespace std;
int main()
```

```
{
    array<int, 5> words{1,2,3,4,5};
    cout << *( words.data()+1);
    return 0;
}
```

（2）访问array容器中多个元素

array容器提供的size()函数能够返回容器中元素的个数（函数返回值为size_t类型）。

```
double total = 0;
for(size_t i = 0 ; i < values.size() ; ++i)
{
    total += values[i];
}
```

size() 函数的存在，为 array 容器提供了标准数组所没有的优势，即能够知道它包含多少元素。

并且，接受数组容器作为参数的函数，只需要通过调用容器的成员函数size()，就能得到元素的个数。除此之外，通过调用array容器的empty()成员函数，即可知道容器中有没有元素（如果容器中没有元素，此函数返回true），如下所示：

```
if(values.empty())
    std::cout << "The container has no elements.\n";
else
    std::cout << "The container has "<< values.size()<<"elements.\n";
```

然而，很少会创建空的array容器，因为当生成一个array容器时，它的元素个数就固定了，而且无法改变，所以生成空array容器的唯一方法是将模板的第二个参数指定为0，但这种情况基本不可能发生。

除了借助 size() 外，对于任何可以使用迭代器的容器，都可以使用基于范围的循环，因此能够更加简便地计算容器中所有元素的和，比如：

```
double total = 0;
for(auto&& value : values)
    total += value;
```

4、vector容器用法总结

4.1 vector的简单介绍

vector作为STL提供的标准容器之一，是经常要使用的，具有重要地位，使用方便。vector又称为向量，vector可以形象的描述为长度可以动态改变的数组，功能和数组较为相似。实际上更专业的描述为：vector是一个多功能的，能够操作多种数据结构和算法的模板类和函数库，vector之所以被认为是一个容器，是因为它能够像容器一样存放各种类型的对象，简单地说，vector是一个能够存放任意类型的动态数组，能够增加和压缩数据。（注：STL的容器从实现的角度讲可以说是类模板class teplate)。

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。

vector和数组的主要区别?

数组：分配的是静态空间，一般分配了就不可以改变，就像我们熟知的定义了一个数组，那么数组的长度就不可以改变，我们也不可以进行越界访问，但是编译器不检查越界，这一点在编程的时候要尤为注意。一般申请的数组长度不能满足我们的要求时，我们需要重新申请大一点的数组，然后把原数组的数据复制过来。

vector：分配的是动态空间，即：我们在声明vector容器时可以不指定容器的大小，vector是随着元素的加入，空间自动扩展的。但是，我们需要肯定vector分配的空间是连续的，也就是支持数组的下标随机访问，实际上vector的实现机制是：预留一部分空间，而且预留空间的大小是按一定比率增长的，如果空间不够用的话，要保证连续，就必须重新new一片空间（默认原始空间两倍），然后将原有元素移动到新空间，同时预留新的空间，最后将原来的那部分空间释放掉。这样预留空间的好处就是不用每次向vector中加元素都重新分配空间。

4.2 vector容器特性

（1）顺序序列：顺序容器中的元素按照严格的线性顺序 排序。可以通过元素在序列中的位置访问对应的元素；

（2）动态数组： 支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算术进行该操作。提供了在序列末尾相对快速地添加/删除元素的操作；

（3）能够感知内存分配（Allocator-aware）： 容器使用一个内存分配器对象来动态地处理它的存储需求。

4.3 vector基本函数使用

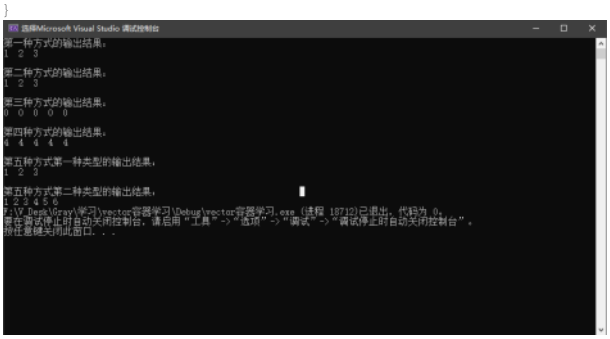
4.3.1构造函数

（1）vector容器的声明方式主要包括以下几种：

```
vector<Elem> v   //创建一个空的vector。
vector <Elem> v1(v)   //复制一个vector。
vector <Elem> v(n)    //创建一个vector，含有n个数据，数据均已缺省构造产生。
vector <Elem> v(n, elem)   //创建一个含有n个elem拷贝的vector。
vector <Elem> v(beg,end)   //创建一个以[beg;end)区间的vector。
```

（2）示例代码：

```
#include<iostream>
#include<vector>
using namespace std;
int main(){
    vector<int>::iterator ite;

    //第一种
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(3);
    cout << "第一种方式的输出结果：" << endl;
    for(ite = v1.begin(); ite != v1.end(); ite ++){
        cout << *ite << "  ";
    }
    cout << endl << endl;

    //第二种方式
    vector<int> v2(v1);
    cout << "第二种方式的输出结果：" << endl;
    for(ite = v2.begin(); ite != v2.end(); ite ++){
        cout << *ite << "  ";
    }
    cout << endl << endl;


    //第三种方式
    vector<int> v3(5);
    cout << "第三种方式的输出结果：" << endl;
    for(ite = v3.begin(); ite != v3.end(); ite ++){
        cout << *ite << "  ";
    }
    cout << endl << endl;

    //第四种方式
    vector<int> v4(5, 4);
    cout << "第四种方式的输出结果：" << endl;
    for(ite = v4.begin(); ite != v4.end(); ite ++){
        cout << *ite << "  ";
    }
    cout << endl << endl;

    //第五种方式
    vector<int> v5(v1.begin(), v1.end());
    cout << "第五种方式第一种类型的输出结果：" << endl;
    for(ite = v5.begin(); ite != v5.end(); ite ++){
        cout << *ite << "  ";
    }
    cout << endl << endl;

     //第六种方式
    int a[] = {1, 2, 3, 4, 5, 6};
    vector<int> v6(a, a+6);
    cout << "第五种方式第二种类型的输出结果：" << endl;
    for(ite = v6.begin(); ite != v6.end(); ite++){
        cout << *ite << " ";
    }
    return 0;
}
```



（3）注意

注意这种：vector c(beg,end)声明方式，创建一个和[beg;end)区间元素相同的vector，一定要注意是左闭右开区间，同时需要说的是，STL中不论是容器还是算法都是采用的这种左闭右开区间的，包括v.end()函数也是返回的vector末端的下位置，相当于int a[n]的a[n]，并不能访问。

4.3.2元素的输入及访问

（1）采用cin和cout

元素的输入及访问可以像操作普通的数组那样，用cin>>进行输入，cout<<a[n]进行输出。

```
#include<iostream>
#include<vector>

using namespace std ;

int main()
{
   vector<int> a(10, 0) ;    //大小为10初值为0的向量a

    //对其中部分元素进行输入
    cin >>a[2] ;
    cin >>a[5] ;
    cin >>a[6] ;

    //全部输出
    int i ;
    for(i=0; i<a.size(); i++)
       cout<<a[i]<<" " ;

    return 0 ;
}
```

（2）采用迭代器

在元素的输出上, 还可以使用遍历器(又称迭代器)进行输出控制。在 vector< int > b(a.begin()，a.begin()+3) ;这种声明形式中, (a.begin()，a.begin()+3) 表示向量起始元素位置到起始元素+3之间的元素位置。 (a.begin()，a.end()) 则表示起始元素和最后一个元素之内的元素位置。

向量元素的位置便成为遍历器, 同时, 向量元素的位置也是一种数据类型, 在向量中遍历器的类型为: vector< int >::iterator。 遍历器不但表示元素位置, 还可以再容器中前后移动。

```
//全部输出
vector<int>::iterator t ;
for(t=a.begin(); t!=a.end(); t++)
    cout<<*t<<" " ;
```

4.3.3 vector中函数

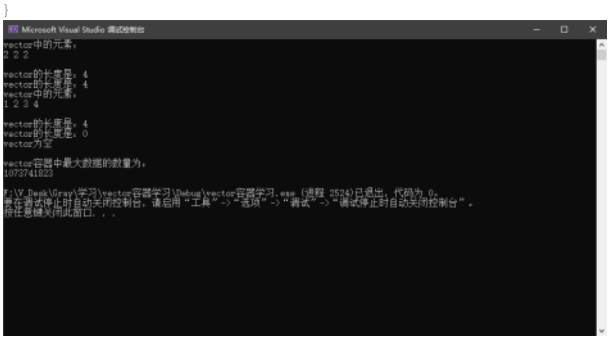
（1）常用函数主要有几下几个:

```
v.assign(beg,end)   //将[beg; end)区间中的数据赋值给v。
v.assign(n,elem)    //将n个elem的拷贝赋值给v。
v.at(idx)           //传回索引idx所指的数据，如果idx越界，抛出out_of_range。也可通过索引直接访问，如:v[0],v[1]。
v.begin()           //传回迭代器第一个数据。
v.capacity()        //返回容器中数据个数。
v.size()            //返回容器中数据个数（与capacity（）有区别）。
v.max_size()        //返回容器中最大数据的数量。
v.clear()           //移除容器中所有数据。
v.empty()           //判断容器是否为空。
v.end()             //指向迭代器中的最后一个数据地址。
v.insert(pos,n,elem)        //在pos位置插入n个elem数据。无返回值。
v.insert(pos,beg,end)       //在pos位置插入在[beg,end)区间的数据。无返回值。
v.erase(pos)                 //删除pos位置的数据，传回下一个数据的位置。
v.erase(beg,end)             //删除[beg,end)区间的数据，传回下一个数据的位置。
v.pop_back()                 //删除最后一个数据。
v.push_back(elem)           //在尾部加入一个数据。
```

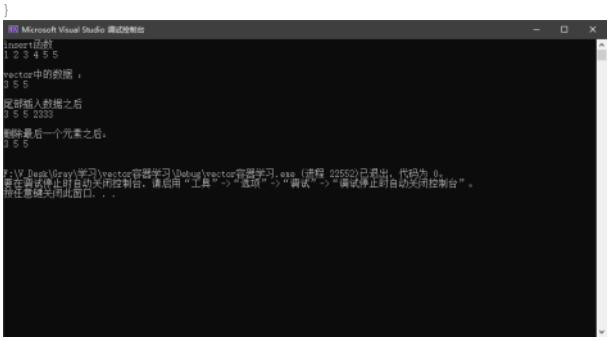
（2）示例代码

```
#include<iostream>
#include<vector>
using namespace std;


int main(){
    vector<int>::iterator ite;


    vector<int> v1;
    int a[] = {1, 2, 3, 4};


    //assign(n,t)
    v1.assign(3,2);
    cout << "vector中的元素："<< endl;
    for(ite=v1.begin(); ite!=v1.end(); ite++){
        cout << *ite << " ";
    }
    cout << endl << endl;


    //assgin(beg, end)
    v1.assign(a, a+4);
    cout << "vector的长度是：" << v1.capacity() << endl;
    cout << "vector的长度是：" << v1.size() << endl;
    cout << "vector中的元素："<< endl;
    for(ite=v1.begin(); ite!=v1.end(); ite++){
        cout << *ite << " ";
    }
```

```
    cout << endl << endl;


    //clear()与empty()
    v1.clear();
    cout << "vector的长度是：" << v1.capacity() << endl;
    cout << "vector的长度是：" << v1.size() << endl;


    if(v1.empty()){
        cout << "vector为空" << endl << endl;;
    }

    cout << "vector容器中最大数据的数量为：" << endl;
    cout << v1.max_size() << endl;
    return 0;
}
```




```
#include<iostream>
#include<vector>
using namespace std;


int main(){
    int a[] = {1, 2, 3, 4};
    vector<int> v1;
    vector<int>::iterator ite;

    //insert函数
    v1.insert(v1.begin(), a, a+4);//新增1 2 3 4
    v1.insert(v1.begin()+4, 2, 5);//新增 5 5
    cout << "insert函数" << endl;
    for(ite = v1.begin(); ite != v1.end(); ite++){
        cout << *ite << " ";
    }
    cout << endl << endl;

    //演示erase函数
    v1.erase(v1.begin(), v1.begin()+2);//删除1 2
    v1.erase(v1.begin()+1);//删除4
    cout << "vector中的数据 ： " << endl;
    for(ite = v1.begin() ; ite != v1.end(); ite++){
        cout << *ite << " ";
    }
    cout << endl << endl;

    //演示push_back函数
    v1.push_back(2333);
    cout << "尾部插入数据之后" << endl;
    for(ite = v1.begin() ; ite != v1.end(); ite++){
        cout << *ite << " ";
    }
    cout << endl << endl;

    //演示pop_back函数
    v1.pop_back();
    cout << "删除最后一个元素之后：" << endl;
    for(ite = v1.begin() ; ite != v1.end(); ite++){
        cout << *ite << " ";
    }
    cout<<endl << endl;

    return 0;
}
```



（3）不常用函数

```
v.resize(num)           //重新指定队列的长度。
c.rbegin()              //传回一个逆向队列的第一个数据。
c.rend()                //传回一个逆向队列的最后一个数据的下一个位置。
c.front()               //传回地一个数据。
c.back()                //传回最后一个数据，不检查这个数据是否存在。
c1.swap(c2)             //将c1和c2元素互换。
```

```
swap(c1,c2)              //同上操作。
```

（4）注意

Ⅰ.vector内存申请测试

在vector尾部循环插入一个数据，同时观察vector的capacity、size、vector首地址，看变化趋势：

```
#include <iostream>
#include <vector> //vector需要包含的头文件
using namespace std;


int main()
{
    vector<int> v1;
    cout<<"初始:cap:"<<v1.capacity()<<",size:"<<v1.size()<<endl;
    for(int i=0;i<65;i++)
    {
        v1.push_back(i); //尾部插入一个数据
        cout<<"cap:"<<v1.capacity()<<",size:"<<v1.size()<<",vector_add:"<<&v1[0]<<endl;
    }
    return 0;
}
```

结果：

```
root@host:/home/LinuxShare/007.STL# g++ 04vecto.cpp
root@host:/home/LinuxShare/007.STL# ./a.out
初始:cap:0,size:0
cap:1,size:1,vector_add:0x55f450486280
cap:2,size:2,vector_add:0x55f4504862a0
cap:4,size:3,vector_add:0x55f450486280
cap:4,size:4,vector_add:0x55f450486280
cap:8,size:5,vector_add:0x55f4504862c0
cap:8,size:6,vector_add:0x55f4504862c0
cap:8,size:7,vector_add:0x55f4504862c0
cap:8,size:8,vector_add:0x55f4504862c0
cap:16,size:9,vector_add:0x55f4504862f0
cap:16,size:10,vector_add:0x55f4504862f0
cap:16,size:11,vector_add:0x55f4504862f0
cap:16,size:12,vector_add:0x55f4504862f0
cap:16,size:13,vector_add:0x55f4504862f0
cap:16,size:14,vector_add:0x55f4504862f0
cap:16,size:15,vector_add:0x55f4504862f0
cap:16,size:16,vector_add:0x55f4504862f0
cap:32,size:17,vector_add:0x55f450486340
cap:32,size:18,vector_add:0x55f450486340
cap:32,size:19,vector_add:0x55f450486340
cap:32,size:20,vector_add:0x55f450486340
cap:32,size:21,vector_add:0x55f450486340
cap:32,size:22,vector_add:0x55f450486340
cap:32,size:23,vector_add:0x55f450486340
cap:32,size:24,vector_add:0x55f450486340
cap:32,size:25,vector_add:0x55f450486340
cap:32,size:26,vector_add:0x55f450486340
cap:32,size:27,vector_add:0x55f450486340
cap:32,size:28,vector_add:0x55f450486340
cap:32,size:29,vector_add:0x55f450486340
cap:32,size:30,vector_add:0x55f450486340
cap:32,size:31,vector_add:0x55f450486340
cap:32,size:32,vector_add:0x55f450486340
cap:64,size:33,vector_add:0x55f4504863d0
cap:64,size:34,vector_add:0x55f4504863d0
cap:64,size:35,vector_add:0x55f4504863d0
cap:64,size:36,vector_add:0x55f4504863d0
cap:64,size:37,vector_add:0x55f4504863d0
cap:64,size:38,vector_add:0x55f4504863d0
cap:64,size:39,vector_add:0x55f4504863d0
cap:64,size:40,vector_add:0x55f4504863d0
cap:64,size:41,vector_add:0x55f4504863d0
cap:64,size:42,vector_add:0x55f4504863d0
cap:64,size:43,vector_add:0x55f4504863d0
cap:64,size:44,vector_add:0x55f4504863d0
cap:64,size:45,vector_add:0x55f4504863d0
cap:64,size:46,vector_add:0x55f4504863d0
cap:64,size:47,vector_add:0x55f4504863d0
cap:64,size:48,vector_add:0x55f4504863d0
cap:64,size:49,vector_add:0x55f4504863d0
cap:64,size:50,vector_add:0x55f4504863d0
cap:64,size:51,vector_add:0x55f4504863d0
cap:64,size:52,vector_add:0x55f4504863d0
cap:64,size:53,vector_add:0x55f4504863d0
cap:64,size:54,vector_add:0x55f4504863d0
cap:64,size:55,vector_add:0x55f4504863d0
cap:64,size:56,vector_add:0x55f4504863d0
cap:64,size:57,vector_add:0x55f4504863d0
cap:64,size:58,vector_add:0x55f4504863d0
cap:64,size:59,vector_add:0x55f4504863d0
cap:64,size:60,vector_add:0x55f4504863d0
cap:64,size:61,vector_add:0x55f4504863d0
cap:64,size:62,vector_add:0x55f4504863d0
cap:64,size:63,vector_add:0x55f4504863d0
cap:64,size:64,vector_add:0x55f4504863d0
cap:128,size:65,vector_add:0x55f4504864e0
```

结果分析：

vector初始capacity和size都是0；在一个个向尾部添加数据后，当vector空间不足以存储数据时，申请更大的空间以存储新插入的数据。申请空间的总大小为当前空间大小的两倍。

当申请更大空间时，首地址发生变化：说明存储位置是全新的，不是在原来的数组尾部地址后面扩展（无法确定尾部后面地址空间是否已经被其他数据占用，干脆重新申请一块两倍大小的空间，让系统自己去分配地址）。

Ⅱ.resize()减小vector后，capacity是否减小？

```
#include <iostream>
#include <vector> //vector需要包含的头文件
using namespace std;


int main()
{
    vector<int> v;
    for(int i=0;i<10000;i++)
    {
        v.push_back(i);
    }
    cout<<"resize前：capacity:"<<v.capacity()<<",size:"<<v.size()<<endl;
    v.resize(10);
    cout<<"resize后：capacity:"<<v.capacity()<<",size:"<<v.size()<<endl;

    return 0;
}
```

结果：

```
root@host:/home/LinuxShare/007.STL# g++ 06vector.cpp
root@host:/home/LinuxShare/007.STL# ./a.out
resize前：capacity:16384,size:10000
resize后：capacity:16384,size:10
```

结果分析：

resize减少了vector元素个数，但是capacity并没有随着减小，占用了不必要的空间。

Ⅲ.巧用swap收缩vector空间

```
#include <iostream>
#include <vector> //vector需要包含的头文件
using namespace std;


int main()
{
    vector<int> v;
    for(int i=0;i<10000;i++)
    {
        v.push_back(i);
    }
    cout<<"resize前：capacity:"<<v.capacity()<<",size:"<<v.size()<<endl;
    v.resize(10);
    cout<<"resize后：capacity:"<<v.capacity()<<",size:"<<v.size()<<endl;
    //收缩空间,交换
    vector<int>(v).swap(v);
    cout<<"swap后：capacity:"<<v.capacity()<<",size:"<<v.size()<<endl;
    return 0;
}
```

结果：

```
root@host:/home/LinuxShare/007.STL# g++ 06vector.cpp
root@host:/home/LinuxShare/007.STL# ./a.out
resize前：capacity:16384,size:10000
resize后：capacity:16384,size:10
swap后：capacity:10,size:10
```

结果分析：

vector<int>声明的无名对象，使用v去初始化无名对象，然后对无名对象调用swap()函数，和v互换。无名对象使用完后自动释放。

Ⅳ.上例拆分写法

```
#include <iostream>
#include <vector> //vector需要包含的头文件
using namespace std;


int main()
{
    vector<int> v;
    for(int i=0;i<10000;i++)
    {
        v.push_back(i);
    }
    cout<<"resize前：capacity:"<<v.capacity()<<",size:"<<v.size()<<endl;
    v.resize(10);
    cout<<"resize后：capacity:"<<v.capacity()<<",size:"<<v.size()<<endl;
    //收缩空间
    vector<int> v_tmp(v);
    cout<<"swap前：v_tmp.capacity:"<<v_tmp.capacity()<<",v_tmp.size:"<<v_tmp.size()<<endl;
    v_tmp.swap(v);
    cout<<"swap后：capacity:"<<v.capacity()<<",size:"<<v.size()<<endl;
    return 0;
}
```

结果：

```
root@host:/home/LinuxShare/007.STL# g++ 06vector.cpp
root@host:/home/LinuxShare/007.STL# ./a.out
resize前：capacity:16384,size:10000
resize后：capacity:16384,size:10
swap前：v_tmp.capacity:10,v_tmp.size:10
swap后：capacity:10,size:10
```

Ⅴ.使用reserve预留空间提高效率

不使用reserve：

```
#include <iostream>
#include <vector> //vector需要包含的头文件
using namespace std;
int main()
{
    vector<int> v;
    int* v_add = NULL;
    int num;
    for(int i=0;i<10000;i++)
    {
        v.push_back(i);
        if(v_add != &(v[0]))
        {
            cout<<"add["<<i<<"]:"<<&(v[0])<<endl;
            v_add = &v[0];
            num ++;
        }
    }
    cout<<"num:"<<num<<endl;
    return 0;
}
```

结果：

```
root@host:/home/LinuxShare/007.STL# g++ 07vector.cpp
root@host:/home/LinuxShare/007.STL# ./a.out
add[0]:0x564d1ec4ae70
add[1]:0x564d1ec4b2a0
add[2]:0x564d1ec4ae70
add[4]:0x564d1ec4b2c0
add[8]:0x564d1ec4b2f0
add[16]:0x564d1ec4b340
add[32]:0x564d1ec4b3d0
add[64]:0x564d1ec4b4e0
add[128]:0x564d1ec4b6f0
add[256]:0x564d1ec4bb00
add[512]:0x564d1ec4c310
add[1024]:0x564d1ec4d320
add[2048]:0x564d1ec4f330
add[4096]:0x564d1ec53340
add[8192]:0x564d1ec5b350
num:15
```

使用reserve：

```
#include <iostream>
#include <vector> //vector需要包含的头文件
using namespace std;


int main()
{
    vector<int> v;
    int* v_add = NULL;
    int num;
    v.reserve(10000);
    for(int i=0;i<10000;i++)
    {
        v.push_back(i);
        if(v_add != &(v[0]))
        {
            cout<<"add["<<i<<"]:"<<&v[0]<<endl;
            v_add = &v[0];
            num ++;
        }
    }
    cout<<"num:"<<num<<endl;


    return 0;
}
```

结果：

```
root@host:/home/LinuxShare/007.STL# g++ 07vector.cpp
root@host:/home/LinuxShare/007.STL# ./a.out
add[0]:0x556c13abde70
num:1
```

4.3.4 vector中二维数组定义方法

（1）方法一

```
#include <string.h>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    int N = 5, M = 6;
    vector<vector<int> > obj(N); //定义二维动态数组大小5行
    for (int i = 0; i < obj.size(); i++)//动态二维数组为5行6列，值全为0
    {
        obj[i].resize(M);
    }
    for (int i = 0; i < obj.size(); i++)//输出二维动态数组
    {
        for (int j = 0;j < obj[i].size();j++)
        {
            cout << obj[i][j] << " ";
        }
```

```
        cout << "\n";
    }
    return 0;
}
```

（2）方法二

```
#include <string.h>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    int N = 5, M = 6;
    vector<vector<int> > obj(N, vector<int>(M)); //定义二维动态数组5行6列
    for (int i = 0; i < obj.size(); i++)//输出二维动态数组
    {
        for (int j = 0;j < obj[i].size();j++)
        {
            cout << obj[i][j] << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

5、deque容器用法总结

5.1 deque的简单介绍

（1）STL提供3种顺序容器：vector、list、deque。vector和deque都是基于数组的，list实现链表数据结构。

（2）deque 是 double-ended queue 的缩写，又称双端队列容器。 deque类在一个容器中提供了vector和list的许多好处。

（3）deque类能利用下标提供有效的索引访问，可以像vector一样读取与修改元素，还像list一样能有效地在前面和后面进行插入和删除操作。

（4）需要增加deque的存储空间时，可以在内存块中deque两端进行分配，通常保存为这些块的指针数组。

（5）对deque分配存储块之后，有些版本要删除deque时才释放这个块，这样使deque比重复分配、释放和再分配内存块时更加有效，但也更浪费内存。

（6）deque利用非连续内存布局，因此deque迭代器比vector和基于指针的数组中用于迭代的指针更加智能化。

（7） deque支持随机访问迭代器。

deque与vector的相同与不同之处：

相同：

（1）deque 容器也擅长在序列尾部添加或删除元素（时间复杂度为0(1)），而不擅长在序列中间添加或删除元素。

（2）deque 容器也可以根据需要修改自身的容量和大小。

不同： deque 还擅长在序列头部添加或删除元素，所耗费的时间复杂度也为常数阶O(1)。并且更重要的一点是，deque 容器中存储元素并不能保证所有元素都存储到连续的内存空间中。当需要向序列两端频繁地添加或删除元素时，应首选deque容器。

5.2 deque的函数使用

5.2.1创建deque容器

创建deque容器，根据不同的实际场景，可选择使用如下几种方式：

（1）创建一个没有任何元素的空deque容器

```
std::deque<int> d;
```

与空array容器不同，空的deque容器在创建之后可以做添加或删除元素的操作，因此这种简单创建deque容器的方式比较常见。

（2）此行代码创建一个具有 10 个元素（默认都为 0）的 deque 容器。

```
std::deque<int> d(10);
```

此行代码创建一个具有 10 个元素（默认都为 0）的 deque 容器。

（3）创建一个具有n个元素的deque容器，并为每个元素都指定初始值。

```
std::deque<int> d(10, 5)
```

此行代码创建了一个包含10个元素（值都为5）的deque容器。

（4）在已有deque容器的情况下，可以通过拷贝该容器创建一个新的deque容器。

```
std::deque<int> d1(5);
std::deque<int> d2(d1);
```

注意，采用此方式，必须保证新旧容器存储的元素类型一致。

（5）通过拷贝其他类型容器中指定区域的元素（也可以是普通数组）创建一个容器。

```
//拷贝普通数组，创建deque容器
int a[] = { 1,2,3,4,5 };
std::deque<int>d(a, a + 5);
//适用于所有类型的容器
std::array<int, 5>arr{ 11,12,13,14,15 };
std::deque<int>d(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}
```

总结

```
//声明list类的实例list1，存放int值，生成了一个长度为0的空list，容量capacity为0
    deque<int>de;


//声明一个list类型的二维链表list1
    deque<deque<int>>de;


//声明一个list类型的一维链表list1,并将{1,2,3,4}赋值给list1
    deque<int>de = { 1,2,3,4 };


//生成一个链表list1，将list2复制给list1
    deque<int>de = de1;
或 deque<int>de(de1);
或  deque<int>de(de1.begin(), de1.end());


//生成一个链表list1，将数组a的a[0]到a[4]的内容复制给list1
    deque<int>de(a, a + 5);
或  deque<int>de(&a[0], &a[4]);


//生成一个链表list1，将大小设为10，且每个元素都为设置为0(默认)
    deque<int>de(10);


//生成一个链表list1，将大小设为10，且每个元素都为设置为5
    deque<int>de(10, 5);


//将list1清空，然后添加2个元素，每个元素都赋值为10
    de.assign(2, 10);


//声明二维链表list1,将两个一维链表赋值给list1
    deque<deque<int>>de;
    deque<int>de1 = { 1,2,3,4 };
    deque<int>de2 = { 2,4,6,8 };
    de.push_back(de1);
    de.push_back(de2);
```

5.2.2 deque容器的成员函数

函数成员	函数功能
begin()	返回指向容器中第一个元素的迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。
rbegin()	返回指向最后一个元素的迭代器。
rend()	返回指向第一个元素所在位置前一个位置的迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
size()	返回实际元素个数。
max_size()	返回容器所能容纳元素个数的最大值。这通常是一个很大的值，一般是 2^{32}-1，我们很少会用到这个函数。
resize()	改变实际元素的个数。
empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
shrink_to_fit()	将内存减少到等于当前元素实际所使用的大小。
at()	使用经过边界检查的索引访问元素。
front()	返回第一个元素的引用。

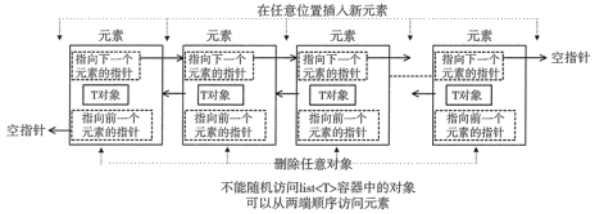
back()	返回最后一个元素的引用。
assign()	用新元素替换原有内容。
push_back()	在序列的尾部添加一个元素。
push_front()	在序列的头部添加一个元素。
pop_back()	移除容器尾部的元素。
pop_front()	移除容器头部的元素。
insert()	在指定的位置插入一个或多个元素。
erase()	移除一个元素或一段元素。
clear()	移出所有的元素，容器大小变为 0。
swap()	交换两个容器的所有元素。
emplace()	在指定的位置直接生成一个元素。
emplace_front()	在容器头部生成一个元素。和 push_front() 的区别是，该函数直接在容器头部构造元素，省去了复制移动元素的过程。
emplace_back()	在容器尾部生成一个元素。和 push_back() 的区别是，该函数直接在容器尾部构造元素，省去了复制移动元素的过程。

```
#include<iostream>
#include<deque>
#include<string>
using namespace std;
int main() {
	deque<int> d;
	d.push_back(10);
	d.push_back(5);
	d.push_front(3);
	d.push_back(15);
	d.push_front(6);
	d.push_back(12);
	d.push_front(16);
	d.push_back(7);
	d.push_back(4);
	d.push_back(2);
	deque<int>::iterator ite;
	cout << "原始序列为：" << endl;
	for (ite = d.begin(); ite != d.end(); ite++) {
		cout << *ite << endl;
	}
	cout << "size为：" << d.size() << endl;
	cout << "最大size为：" << d.max_size() << endl;
	cout << "第一个元素为：" << d.front() << endl;
	cout << "最后一个元素为：" << *(d.end() - 1) << endl;
	d.resize(5);
	cout << "resize后序列为：" << endl;
	for (ite = d.begin(); ite != d.end(); ite++) {
		cout << *ite << endl;
	}
	return 0;
}
```

6、list容器用法总结

6.1 list的简单介绍

list容器，又称双向链表容器，即该容器的底层是以双向链表的形式实现的。这意味着，list容器中的元素可以分散存储在内存空间里，而不是必须存储在一整块连续的内存空间中。



可以看出，list容器中各个元素的前后顺序是靠指针来维系的，每个元素都配备了两个指针，分别指向它的前一个元素和后一个元素。其中第一个元素的前向指针总为null，因为它前面没有元素；同样，尾部元素的后向指针也总为null。

基于这样的存储结构，list容器具有一些其他容器（array、vector和deque）所不具备的优势，即它可以在序列已知的任何

位置快速插入或删除元素（时间复杂度为O(1)）。并且在list容器中移动元素，也比其他容器效率高。
使用list容器的缺点是，它不能像array和vector那样，通过位置直接访问元素。举个例子，如果要访问list容器中的第6个元素，它不支持容器对象名[6]这种语法格式，正确的做法是从容器中第一个元素或最后一个元素开始b遍历容器，指导找到该位置。
实际场景中，如果需要对序列进行大量添加或删除元素的操作，而直接访问元素的需求却很少，这种情况建议使用list容器存储序列。

6.2 list的函数使用

6.2.1创建list容器

根据不同的使用场景，有以下五种创建list容器的方式供选择：
（1）创建一个没有任何元素的空list容器。

```
std::list<int> values;
```

和空array不同， 空的 list 容器在创建之后仍可以添加元素，因此创建 list 容器的方式很常用。
（2）创建一个包含n个元素的list容器。

```
std::list<int> values(10);
```

通过此方式创建 values 容器，其中包含 10 个元素，每个元素的值都为相应类型的默认值（int类型的默认值为 0）。
（3）创建一个包含n个元素的list容器 并为每个元素指定初始值。

```
std::list<int> values(10, 5);
```

创建了一个包含 10 个元素并且值都为 5 个 values 容器。
（4）在已有list容器的情况下，通过拷贝该容器可以创建新的list容器。

```
std::list<int> value1(10);
std::list<int> value2(value1);
```

注意，采用此方式，必须保证新旧容器存储的元素类型一致。
（5）通过拷贝其他类型容器（或者普通数组）种指定区域内的元素，可以创建新的list容器。

```
//拷贝普通数组，创建list容器
int a[] = { 1,2,3,4,5 };
std::list<int> values(a, a+5);
//拷贝其它类型的容器，创建 list 容器
std::array<int, 5>arr{ 11,12,13,14,15 };
std::list<int>values(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}
```

6.2.2 list容器的成员函数

函数成员	函数功能
begin()	返回指向容器中第一个元素的双向迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的双向迭代器。
rbegin()	返回指向最后一个元素的反向双向迭代器。
rend()	返回指向第一个元素所在位置前一个位置的反向双向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
size()	返回实际元素个数。
max_size()	返回容器所能容纳元素个数的最大值。这通常是一个很大的值，一般是 2^{32}-1，我们很少会用到这个函数。
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
assign()	用新元素替换容器中原有内容。
emplace_front(	在容器头部生成一个元素。该函数和 push_front() 的功能相同，但效率更高。

)	
push_front()	在容器头部插入一个元素。
pop_front()	删除容器头部的一个元素。
emplace_back()	在容器尾部直接生成一个元素。该函数和 push_back() 的功能相同，但效率更高。
push_back()	在容器尾部插入一个元素。
pop_back()	删除容器尾部的一个元素。
emplace()	在容器中的指定位置插入元素。该函数和 insert() 功能相同，但效率更高。
insert()	在容器中的指定位置插入元素。
erase()	删除容器中一个或某区域内的元素。
swap()	交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。
resize()	调整容器的大小。
clear()	删除容器存储的所有元素。
splice()	将一个 list 容器中的元素插入到另一个容器的指定位置。
remove(val)	删除容器中所有等于 val 的元素。
remove_if()	删除容器中满足条件的元素。
unique()	删除容器中相邻的重复元素，只保留一个。
merge()	合并两个事先已排好序的 list 容器，并且合并之后的 list 容器依然是有序的。
sort()	通过更改容器中元素的位置，将它们进行排序。
reverse()	反转容器中元素的顺序。

（1）list赋值和交换

```
#include<iostream>
#include<list>
using namespace std;
void test() {
    list<int> l;         //默认构造
    l.push_back(1);      //向容器尾部插入数据
    l.push_back(2);
    l.push_back(3);
    l.push_back(4);
    l.push_back(5);
    list<int> l1;
    l1.assign(l.begin(), l.end());         //将容器l从头到尾的数据拷贝给容器l1
    list<int> l2(9, 2);        //将9个2拷贝赋值给容器l2
    list<int> l3;
    l3 = l;         //重载=操作符
    l3.swap(l2);        //交换两个容器中的元素
    list<int>::iterator ite;
    for (ite = l3.begin(); ite != l3.end(); ite++) {
        cout << *ite << endl;
    }
}
int main() {
    test();
    return 0;
}
```

（2）list大小操作

```
void test() {
    list<int> l;
    l.push_back(1);
    l.push_back(2);
    l.push_back(3);
    l.push_back(4);
    l.push_back(5);
    if (l.empty()) {          //判断容器是否为空
        cout << "l为空" << endl;
    }
    else {
        cout << "l不为空" << endl;
        cout << "l的元素个数为：" << l.size() << endl;
    }
    //l.resize(10);
    l.resize(10,8);
    l.resize(2);}
```

（3）list删除与插入

```
void test() {
    list<int> l;
    l.push_back(1);   //尾部插入元素
```

```
    l.push_back(2);
    l.push_back(3);
    l.push_back(4);
    l.push_front(5);  //头部插入元素
    l.push_front(6);
    l.push_front(7);
    l.push_front(8);      //8 7 6 5 1 2 3 4
    l.pop_back();   //删除尾部数据  8 7 6 5 1 2 3
    l.pop_front();   //删除头部数据   7 6 5 1 2 3
    l.insert(l.begin(), 10);    //迭代器指向容器头部，在此位置插入元素10       10 7 6 5 1 2 3
    list<int>::iterator it = l.begin();
    l.insert(++it, 3, 11);    //迭代器指向容器头部的下一个元素位置，在此位置插入3个11     10 11 11 11 7 6 5 1 2 3
    l.erase(l.begin());     //迭代器指向容器头部，删除迭代器指向的元素  11 11 11 7 6 5 1 2 3
    l.unique();   //删除重复的元素，同一个元素只留下一个   11 7 6 5 1 2 3
    l.push_back(11);
    l.push_back(11);
    l.push_back(11);    11 7 6 5 1 2 3 11 11 11
    l.remove(11);    //删除容器中所有的11   7 6 5 1 2 3
    l.clear();       //清空容器}
```

（4）list数据存取

```
void test() {
    list<int> l;
    l.push_back(1);
    l.push_back(2);
    l.push_back(3);
    l.push_back(4);
    l.push_back(5);
    cout << "容器1的第一个元素是：" << l.front() << endl;      //返回1
    cout << "容器1的最后一个元素是：" << l.back() << endl;    //返回5}
```

（5）list反转和排序

```
void printList(const list<int> &l) {        //打印容器中的元素
    for (list<int>::const_iterator it = l.begin(); it != l.end(); it++) {     //迭代器可以自增自减
        cout << *it << " ";
    }
    cout << endl;}
void test1() {
    list<int> l1;
    l1.push_back(2);
    l1.push_back(7);
    l1.push_back(5);
    l1.push_back(1);
    l1.push_back(9);
    printList(l1);              //打印结果为：2 7 5 1 9
    l1.reverse();               //反转容器中的元素
    printList(l1);              //打印结果为：9 1 5 7 2
    l1.sort();                  //对容器中的元素进行排序
    printList(l1);              //打印结果为：1 2 5 7 9}
```

（6）list迭代器及用法

与array、vector、deque容器的迭代器相比，list容器迭代器最大的不同之处在于，其配备的迭代器类型为双向迭代器，而不再是随机访问迭代器。这意味着，假设p1和p2都是双向迭代器，则它们支持使用++p1、p1++、p1--、p1++、*p1、p1==p2以及p1!=p2运算符，但不支持以下操作（其中i为整数）：

- p1[i]：不能通过下标访问list容器中指定位置处的元素；
- p1-=i、p1+=i、p1+i、p1-i：双向迭代器不支持使用-=、+=、+、-运算符；
- p1<p2、p1>p2、p1<=p2、p1>=p2：双向迭代器p1、p2不支持使用<、>、<=、>=比较运算符。

```
#include<iostream>
#include<list>
using namespace std;
int main()
{
    //创建list容器
    std::list<char> values{'h','t','t','p',':','/','/','c','.','b','i','a','n','c','h','e','n','g','.','n','e','t'};
    //使用begin()/end()迭代器函数输出list容器中的元素
    for(std::list<char>::iterator it = values.begin(); it != values.end(); ++it){
        std::cout << *it;
    }
    cout << endl;
    //使用 rbegin()/rend()迭代器函数输出 lsit 容器中的元素
    for (std::list<char>::reverse_iterator it = values.rbegin(); it != values.rend();++it) {
        std::cout << *it;
    }
    return 0;
}
```

注意，程序中比较迭代器之间的关系，用的是!=运算符，因为它不支持<等运算符。另外在实际场景中，所有迭代器函数的返回值都可以传给使用auto关键字定义的变量，因为编译器可以自行判断出该迭代器的类型。

值得一提的是，list容器在进行插入（insert()）、接合（splice()）等操作时，都不会造成原有的list迭代器失效，甚至进行删除操作，而只有指向被删除元素的迭代器失效，其他迭代器不受任何影响。

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    //创建 list 容器
    std::list<char> values{'h','t','t','p',':','/','/','c','.','b','i','a','n','c','h','e','n','g','.','n','e','t'};
    //创建 begin 和 end 迭代器
    std::list<char>::iterator begin = values.begin();
    std::list<char>::iterator end = values.end();
    //头部和尾部插入字符 '1'
```

```
    values.insert(begin, '1');
    values.insert(end, '1');
    while (begin != end)
    {
        std::cout << *begin;
        ++begin;
    }
    return 0;
}
```

http://c.biancheng.net1

在进行插入操作之后，仍使用先前创建的迭代器遍历容器，虽然程序不会出错，但由于插入位置的不同，可能会遗漏新插入的元素。

（7）list容器中splice函数的用法

和insert()成员方法相比，splice()成员方法的作用对象是其它list容器，其功能是将其它list容器中的元素添加到当前list容器中指定位置处。

语法格式	功能
void splice(iterator position, list& x);	position为迭代器，用于指明插入位置；x为另一个list容器。此格式splice()方法的功能是，将x容器中存储的所有元素全部移动当前list容器中Position指明的位置处。
void splice (iterator position, list& x, iterator i);	position 为迭代器，用于指明插入位置；x 为另一个 list 容器；i 也是一个迭代器，用于指向 x 容器中某个元素。 此格式的 splice() 方法的功能是将 x 容器中 i 指向的元素移动到当前容器中 position 指明的位置处。
void splice (iterator position, list& x, iterator first, iterator last);	position 为迭代器，用于指明插入位置；x 为另一个 list 容器；first 和 last 都是迭代器，[fist,last) 用于指定 x 容器中的某个区域。 此格式的 splice() 方法的功能是将 x 容器 [first, last) 范围内所有的元素移动到当前容器 position 指明的位置处。

list 容器底层使用的是链表存储结构，splice() 成员方法移动元素的方式是，将存储该元素的节点从 list 容器底层的链表中摘除，然后再链接到当前 list 容器底层的链表中。这意味着，当使用 splice() 成员方法将 x 容器中的元素添加到当前容器的同时，该元素会从 x 容器中删除。

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    //创建并初始化 2 个 list 容器
    list<int> mylist1{ 1,2,3,4 }, mylist2{10,20,30};
    list<int>::iterator it = ++mylist1.begin(); //指向 mylist1 容器中的元素 2

    //调用第一种语法格式
    mylist1.splice(it, mylist2); // mylist1: 1 10 20 30 2 3 4
                                 // mylist2:
                                 // it 迭代器仍然指向元素 2，只不过容器变为了 mylist1


    //调用第二种语法格式，将 it 指向的元素 2 移动到 mylist2.begin() 位置处
    mylist2.splice(mylist2.begin(), mylist1, it);   // mylist1: 1 10 20 30 3 4
                                                    // mylist2: 2
                                                    // it 仍然指向元素 2

    //调用第三种语法格式，将 [mylist1.begin(),mylist1.end())范围内的元素移动到 mylist.begin() 位置处
    mylist2.splice(mylist2.begin(), mylist1, mylist1.begin(), mylist1.end());//mylist1:
                                                                             //mylist2:1 10 20 30 3 4 2

    cout << "mylist1 包含 " << mylist1.size() << "个元素" << endl;
    cout << "mylist2 包含 " << mylist2.size() << "个元素" << endl;
    //输出 mylist2 容器中存储的数据
    cout << "mylist2:";
    for (auto iter = mylist2.begin(); iter != mylist2.end(); ++iter) {
        cout << *iter << " ";
    }
    return 0;
}
mylist1 包含 0个元素
mylist2 包含 7个元素
mylist2:1 10 20 30 3 4 2
```

（8）list容器中unique()函数的用法

unique() 函数也有以下 2 种语法格式：

```
void unique()
void unique（BinaryPredicate）//传入一个二元谓词函数
```

以上 2 种格式都能实现去除 list 容器中相邻重复的元素，仅保留一份。但第 2 种格式的优势在于，我们能自定义去重的规则，例如：

```
#include <iostream>
```

```
#include <list>
using namespace std;
//二元谓词函数
bool demo(double first, double second)
{
    return (int(first) == int(second));
}


int main()
{
    list<double> mylist{ 1,1.2,1.2,3,4,4.5,4.6 };
    //删除相邻重复的元素，仅保留一份
    mylist.unique();//{1, 1.2, 3, 4, 4.5, 4.6}


    for (auto it = mylist.begin(); it != mylist.end(); ++it)
        cout << *it << ' ';
    cout << endl;
    //demo 为二元谓词函数，是我们自定义的去重规则
    mylist.unique(demo);


    for (auto it = mylist.begin(); it != mylist.end(); ++it)
        std::cout << *it << ' ';
    return 0;
}
1 1.2 3 4 4.5 4.6
1 3 4
```

除此之外，通过将自定义的谓词函数（不限定参数个数）传给 remove_if() 成员函数，list 容器中能使谓词函数成立的元素都会被删除。举个例子：

```
#include <iostream>
#include <list>
using namespace std;


int main()
{
    std::list<int> mylist{ 15, 36, 7, 17, 20, 39, 4, 1 };
    //删除 mylist 容器中能够使 lamba 表达式成立的所有元素。
    mylist.remove_if([](int value) {return (value < 10); }); //{15 36 17 20 39}


    for (auto it = mylist.begin(); it != mylist.end(); ++it)
        std::cout << ' ' << *it;


    return 0;
}
```

7、forward_list容器用法总结

7.1 forward_list的简单介绍

forward_list 是C++11 新添加的一类容器，其底层实现和 list 容器一样，采用的也是链表结构，只不过 forward_list 使用的是单链表，而 list 使用的是双向链表。 因此，forward_list 容器具有和 list 容器相同的特性，即擅长在序列的任何位置进行插入元素或删除元素的操作，但对于访问存储的元素，没有其它容器（如 array、vector）的效率高。

另外，由于单链表没有双向链表那样灵活，因此相比 list 容器，forward_list 容器的功能受到了很多限制。比如，由于单链表只能从前向后遍历，而不支持反向遍历，因此 forward_list 容器只提供前向迭代器，而不是双向迭代器。这意味着，forward_list 容器不具有 rbegin()、rend() 之类的成员函数。

那么，既然 forward_list 容器具有和 list 容器相同的特性，list 容器还可以提供更多的功能函数，forward_list 容器有什么存在的必要呢?

forward_list 容器底层使用单链表，也不是一无是处。比如，存储相同个数的同类型元素，单链表耗用的内存空间更少，空间利用率更高，并且对于实现某些操作单链表的执行效率也更高。

7.2 forward_list的函数使用

7.2.1创建forward_list容器

创建 forward_list 容器的方式，大致分为以下 5 种。

(1)创建一个没有任何元素的空forward_list容器。

```
std::forward_list<int> values;
```

(2)创建一个包含n个元素的forward_list容器。

```
std::forward_list<int> values(10);
```

通过此方式创建 values 容器，其中包含 10 个元素，每个元素的值都为相应类型的默认值（int类型的默认值为 0）。

(3)创建一个包含n个元素的forward_list容器，并为每个元素指定初始值。

```
std::forward_list<int> values(10, 5);
```

创建了一个包含 10 个元素并且值都为 5 个 values 容器。

(4) 在已有 forward_list 容器的情况下，通过拷贝该容器可以创建新的 forward_list 容器。

```
std::forward_list<int> value1(10);
std::forward_list<int> value2(value1);
```

注意，采用此方式，必须保证新旧容器存储的元素类型一致。

(5) 通过拷贝其他类型容器（或者普通数组）中指定区域内的元素，可以创建新的 forward_list 容器。

```
//拷贝普通数组，创建forward_list容器
int a[] = { 1,2,3,4,5 };
std::forward_list<int> values(a, a+5);
//拷贝其它类型的容器，创建forward_list容器
std::array<int, 5>arr{ 11,12,13,14,15 };
std::forward_list<int>values(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}
```

7.2.2 forward_list容器的成员函数

函数成员	函数功能
before_begin()	返回一个前向迭代器，其指向容器中第一个元素之前的位置。
begin()	返回指向容器中第一个元素的前向迭代器。
end()	返回指向容器最后一个元素所在位置后一个位置的前向迭代器。
cbefore_begin()	和 before_begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
max_size()	返回容器所能容纳元素个数的最大值。这通常是一个很大的值，一般是 $2^{32}-1$，我们很少会用到这个函数。
front()	返回第一个元素的引用。
back()	返回最后一个元素的引用。
assign()	用新元素替换容器中原有内容。
emplace_front()	在容器头部生成一个元素。该函数和 push_front() 的功能相同，但效率更高。
push_front()	在容器头部插入一个元素。
pop_front()	删除容器头部的一个元素。
emplace_after()	在指定位置之后插入一个新元素，并返回一个指向新元素的迭代器。和 insert_after() 的功能相同，但效率更高。
insert_after()	在指定位置之后插入一个新元素，并返回一个指向新元素的迭代器。
erase_after()	删除容器中某个指定位置或区域内的所有元素。
swap()	交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。
resize()	调整容器的大小。
clear()	删除容器存储的所有元素。
splice_after()	将某个 forward_list 容器中指定位置或区域内的元素插入到另一个容器的指定位置之后。
remove(val)	删除容器中所有等于 val 的元素。
remove_if()	删除容器中满足条件的元素。
unique()	删除容器中相邻的重复元素，只保留一个。
merge()	合并两个事先已排好序的 forward_list 容器，并且合并之后的 forward_list 容器依然是有序的。
sort()	通过更改容器中元素的位置，将它们进行排序。
reverse()	反转容器中元素的顺序。

(1) forward_list 容器中是不提供 size() 函数的，但如果想要获取 forward_list 容器中存储元素的个数，可以使用头文

件 <iterator> 中的 distance() 函数。

```
#include <iostream>
#include <forward_list>
#include <iterator>
using namespace std;


int main()
{
    std::forward_list<int> my_words{1,2,3,4};
    int count = std::distance(std::begin(my_words), std::end(my_words));
    cout << count;
    return 0;
}
```

(2) forward_list 容器迭代器的移动除了使用 ++ 运算符单步移动，可以使用 advance() 函数。

```
#include <iostream>
#include <forward_list>
using namespace std;


int main()
{
    std::forward_list<int> values{1,2,3,4};
    auto it = values.begin();
    advance(it, 2);
    while (it!=values.end())
    {
        cout << *it << " ";
        ++it;
    }
    return 0;
}
```

8、map容器用法总结

8.1 map的简单介绍

作为关联式容器的一种，map 容器存储的都是 pair 对象，也就是用 pair 类模板创建的键值对。其中，各个键值对的键和值可以是任意数据类型，包括C++基本数据类型（int、double 等）、使用结构体或类自定义的类型。 通常情况下，map 容器中存储的各个键值对都选用 string 字符串作为键的类型。

与此同时，在使用 map 容器存储多个键值对时，该容器会自动根据各键值对的键的大小，按照既定的规则进行排序。默认情况下，map 容器选用std::less<T>排序规则（其中 T 表示键的数据类型），其会根据键的大小对所有键值对做升序排序。当然，根据实际情况的需要，我们可以手动指定 map 容器的排序规则，既可以选用STL标准库中提供的其它排序规则（比如std::greater<T>），也可以自定义排序规则。

另外需要注意的是，使用 map 容器存储的各个键值对，键的值既不能重复也不能被修改。换句话说，map 容器中存储的各个键值对不仅键的值独一无二，键的类型也会用 const 修饰，这意味着只要键值对被存储到 map 容器中，其键的值将不能再做任何修改。

8.2 map的函数使用

8.2.1创建map容器

（1）通过调用map容器类的默认构造函数，可以创建出一个空的map容器。

```
std::map<std::string, int>myMap;
```

通过此方式创建出的 myMap 容器，初始状态下是空的，即没有存储任何键值对。鉴于空 map 容器可以根据需要随时添加新的键值对，因此创建空 map 容器是比较常用的。

（2）初始化map容器。

```
std::map<std::string, int>myMap{ {"C语言教程",10},{"STL教程",20} };
```

通过这种方式，myMap 容器在初始状态下，就包含有 2 个键值对。再次强调，map 容器中存储的键值对，其本质都是 pair 类模板创建的 pair 对象。因此，下面程序也可以创建出一模一样的 myMap 容器：

```
std::map<std::string, int>myMap{std::make_pair("C语言教程",10),std::make_pair("STL教程",20)};
```

（3） 在某些场景中，可以利用先前已创建好的 map 容器，再创建一个新的 map 容器。

```
std::map<std::string, int>newMap(myMap);
```

由此，通过调用 map 容器的拷贝（复制）构造函数，即可成功创建一个和 myMap 完全一样的 newMap 容器。

C++ 11 标准中，还为 map 容器增添了移动构造函数。当有临时的 map 对象作为参数，传递给要初始化的 map 容器时，此时就会调用移动构造函数。

```
//创建一个会返回临时map对象的函数
std::map<std::string, int> disMap(){
    std::map<std::string, int> tempMap{{"C语言教程", 10}, {"STL教程", 20}};
    return tempMap;
}
//调用map类模板的移动构造函数创建newMap容器
std::map<std::string, int> newMap(disMap());
```

（4）map 类模板还支持取已建 map 容器中指定区域内的键值对，创建并初始化新的 map 容器。

```
std::map<std::string, int>myMap{ {"C语言教程",10},{"STL教程",20} };
std::map<std::string, int>newMap(++myMap.begin(), myMap.end());
```

这里，通过调用 map 容器的双向迭代器，实现了在创建 newMap 容器的同时，将其初始化为包含一个 {"STL教程", 20} 键值对的容器。

（5）当然，在以上几种创建 map 容器的基础上，我们都可以手动修改 map 容器的排序规则。默认情况下，map 容器调用 std::less<T> 规则，根据容器内各键值对的键的大小，对所有键值对做升序排序。

```
std::map<std::string, int>myMap{ {"C语言教程",10},{"STL教程",20} };
std::map<std::string, int, std::less<std::string> >myMap{ {"C语言教程",10},{"STL教程",20} };
```

以上两行代码是等效的，也可以 手动修改了 myMap 容器的排序规则，令其作降序排序。

```
std::map<std::string, int, std::greater<std::string> >myMap{ {"C语言教程",10},{"STL教程",20} };
```

8.2.2 map容器的成员函数

成员函数	功能
begin()	返回指向容器中第一个（注意，是已排好序的第一个）键值对的双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
end()	返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
rbegin()	返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。
rend()	返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
find(key)	在 map 容器中查找键为 key 的键值对，如果成功找到，则返回指向该键值对的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
lower_bound(key)	返回一个指向当前 map 容器中第一个大于或等于 key 的键值对的双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
upper_bound(key)	返回一个指向当前 map 容器中第一个大于 key 的键值对的迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
equal_range(key)	该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower_bound() 方法的返回值等价，pair.second 和 upper_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的键为 key 的键值对（map 容器键值对唯一，因此该范围最多包含一个键值对）。
empty()	若容器为空，则返回 true；否则 false。
size()	返回当前 map 容器中存有键值对的个数。
max_size()	返回 map 容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。
operator[]	map容器重载了 [] 运算符，只要知道 map 容器中某个键值对的键的值，就可以向获取数组中元素那样，通过键直接获取对应的值。
at(key)	找到 map 容器中 key 键对应的值，如果找不到，该函数会引发 out_of_range 异常。
insert()	向 map 容器中插入键值对。
erase()	删除 map 容器指定位置、指定键（key）值或者指定区域内的键值对。
swap()	交换 2 个 map 容器中存储的键值对，这意味着，操作的 2 个键值对的类型必须相同。
clear()	清空 map 容器中所有的键值对，即使 map 容器的 size() 为 0。

emplace()	在当前 map 容器中的指定位置处构造新键值对。其效果和插入键值对一样，但效率更高。
emplace_hint()	在本质上和 emplace() 在 map 容器中构造新键值对的方式是一样的，不同之处在于，使用者必须为该方法提供一个指示键值对生成位置的迭代器，并作为该方法的第一个参数。
count(key)	在当前 map 容器中，查找键为 key 的键值对的个数并返回。注意，由于 map 容器中各键值对的键的值是唯一的，因此该函数的返回值最大为 1。

（1）map容器迭代器用法

C++STL标准库为 map 容器配备的是双向迭代器（bidirectional iterator）。这意味着，map 容器迭代器只能进行 ++p、p++、--p、p--、*p 操作，并且迭代器之间只能使用 == 或者 != 运算符进行比较。

Ⅰ.begin()/end()

```
#include <iostream>
#include <map>      // pair
#include <string>       // string
using namespace std;


int main() {
    //创建并初始化 map 容器
    std::map<std::string, std::string>myMap{ {"STL教程","http://c.biancheng.net/stl/"},{"C语言教程","http://c.biancheng.net/c/"} };


    //调用 begin()/end() 组合，遍历 map 容器
    for (auto iter = myMap.begin(); iter != myMap.end(); ++iter) {
        cout << iter->first << " " << iter->second << endl;
    }
    return 0;
}
C语言教程 http://c.biancheng.net/c/
STL教程 http://c.biancheng.net/stl/
```

Ⅱ.find()

```
#include <iostream>
#include <map>      // pair
#include <string>       // string
using namespace std;


int main() {
    //创建并初始化 map 容器
    std::map<std::string, std::string>myMap{ {"STL教程","http://c.biancheng.net/stl/"},
                                             {"C语言教程","http://c.biancheng.net/c/"},
                                             {"Java教程","http://c.biancheng.net/java/"} };
    //查找键为 "Java教程" 的键值对
    auto iter = myMap.find("Java教程");
    //从 iter 开始，遍历 map 容器
    for (; iter != myMap.end(); ++iter) {
        cout << iter->first << " " << iter->second << endl;
    }
    return 0;
}
Java教程 http://c.biancheng.net/java/
STL教程 http://c.biancheng.net/stl/
```

Ⅲ.lower_bound(key)/upper_bound(key)

```
#include <iostream>
#include <map>      // pair
#include <string>       // string
using namespace std;


int main() {
    //创建并初始化 map 容器
    std::map<std::string, std::string>myMap{ {"STL教程","http://c.biancheng.net/stl/"},
                                             {"C语言教程","http://c.biancheng.net/c/"},
                                             {"Java教程","http://c.biancheng.net/java/"} };
    //找到第一个键的值不小于 "Java教程" 的键值对
    auto iter = myMap.lower_bound("Java教程");
    cout << "lower：" << iter->first << " " << iter->second << endl;

    //找到第一个键的值大于 "Java教程" 的键值对
    iter = myMap.upper_bound("Java教程");
    cout <<"upper：" << iter->first << " " << iter->second << endl;
    return 0;
}
lower：Java教程 http://c.biancheng.net/java/
upper：STL教程 http://c.biancheng.net/stl/
```

IV.equal_range(key)

```
#include <iostream>
#include <utility>  //pair
#include <map>      // map
#include <string>       // string
using namespace std;


int main() {
    //创建并初始化 map 容器
    std::map<string, string>myMap{ {"STL教程","http://c.biancheng.net/stl/"},
                                   {"C语言教程","http://c.biancheng.net/c/"},
                                   {"Java教程","http://c.biancheng.net/java/"} };
    //创建一个 pair 对象，来接收 equal_range() 的返回值
    pair <std::map<string, string>::iterator, std::map<string, string>::iterator> myPair = myMap.equal_range("C语言教程");
    //通过遍历，输出 myPair 指定范围内的键值对
```

```
    for (auto iter = myPair.first; iter != myPair.second; ++iter) {
        cout << iter->first << " " << iter->second << endl;
    }
    return 0;
}
```
C语言教程 http://c.biancheng.net/c/

（2）map容器获取键对应值的几种方法

Ⅰ.map类容器中对[]运算符进行了重载，这意味着，类似于借助数组下标可以直接访问数组中的元素，通过指定的键，可以轻松获取map容器中该键对应的值。

```
#include <iostream>
#include <map>      // map
#include <string>   // string
using namespace std;


int main() {
    //创建并初始化 map 容器
    std::map<std::string, std::string>myMap{ {"STL教程","http://c.biancheng.net/stl/"},
                                             {"C语言教程","http://c.biancheng.net/c/"},
                                             {"Java教程","http://c.biancheng.net/java/"} };
    string cValue = myMap["C语言教程"];
    cout << cValue << endl;
    return 0;
}
```
http://c.biancheng.net/c/

注意，只有当 map 容器中确实存有包含该指定键的键值对，借助重载的 [ ] 运算符才能成功获取该键对应的值；反之，若当前 map 容器中没有包含该指定键的键值对，则此时使用 [ ] 运算符将不再是访问容器中的元素，而变成了向该 map 容器中增添一个键值对。其中，该键值对的键用 [ ] 运算符中指定的键，其对应的值取决于 map 容器规定键值对中值的数据类型，如果是基本数据类型，则值为 0；如果是 string 类型，其值为 ""，即空字符串（即使用该类型的默认值作为键值对的值）。

```
#include <iostream>
#include <map>      // map
#include <string>   // string
using namespace std;


int main() {
    //创建空 map 容器
    std::map<std::string, int>myMap;
    int cValue = myMap["C语言教程"];
    for (auto i = myMap.begin(); i != myMap.end(); ++i) {
        cout << i->first << " "<< i->second << endl;
    }
    return 0;
}
```
C语言教程 0

显然，对于空的 myMap 容器来说，其内部没有以 "C语言教程" 为键的键值对，这种情况下如果使用 [ ] 运算符获取该键对应的值，其功能就转变成了向该 myMap 容器中添加一个<"C语言教程"，0>键值对（由于 myMap 容器规定各个键值对的值的类型为 int，该类型的默认值为 0）。

实际上，[ ] 运算符确实有“为 map 容器添加新键值对”的功能，但前提是要保证新添加键值对的键和当前 map 容器中已存储的键值对的键都不一样。例如：

```
#include <iostream>
#include <map>      // map
#include <string>   // string
using namespace std;


int main() {
    //创建空 map 容器
    std::map<string, string>myMap;
    myMap["STL教程"]="http://c.biancheng.net/java/";
    myMap["Python教程"] = "http://c.biancheng.net/python/";
    myMap["STL教程"] = "http://c.biancheng.net/stl/";
    for (auto i = myMap.begin(); i != myMap.end(); ++i) {
        cout << i->first << " " << i->second << endl;
    }
    return 0;
}
```
Python教程 http://c.biancheng.net/python/
STL教程 http://c.biancheng.net/stl/

Ⅱ. 除了借助 [ ] 运算符获取 map 容器中指定键对应的值，还可以使用 at() 成员方法。和前一种方法相比，at() 成员方法也需要根据指定的键，才能从容器中找到该键对应的值；不同之处在于，如果在当前容器中查找失败，该方法不会向容器中添加新的键值对，而是直接抛出 out_of_range 异常。

```
#include <iostream>
#include <map>      // map
#include <string>   // string
using namespace std;


int main() {
    //创建并初始化 map 容器
    std::map<std::string, std::string>myMap{ {"STL教程","http://c.biancheng.net/stl/"},
                                             {"C语言教程","http://c.biancheng.net/c/"},
                                             {"Java教程","http://c.biancheng.net/java/"} };
```

```
    cout << myMap.at("C语言教程") << endl;
    //下面一行代码会引发 out_of_range 异常
    //cout << myMap.at("Python教程") << endl;
    return 0;
}
```
http://c.biancheng.net/c/

除了可以直接获取指定键对应的值之外，还可以借助 find() 成员方法间接实现此目的。和以上 2 种方式不同的是，该方法返回的是一个迭代器，即如果查找成功，该迭代器指向查找到的键值对；反之，则指向 map 容器最后一个键值对之后的位置（和 end() 成功方法返回的迭代器一样）。

```
#include <iostream>
#include <map>      // map
#include <string>   // string
using namespace std;


int main() {
    //创建并初始化 map 容器
    std::map<std::string, std::string>myMap{ {"STL教程","http://c.biancheng.net/stl/"},
                                             {"C语言教程","http://c.biancheng.net/c/"},
                                             {"Java教程","http://c.biancheng.net/java/"} };


    map< std::string, std::string >::iterator myIter = myMap.find("C语言教程");
    cout << myIter->first << " " << myIter->second << endl;
    return 0;
}
```
C语言教程 http://c.biancheng.net/c/

注意，此程序中如果 find() 查找失败，会导致第 13 行代码运行出错。因为当 find() 方法查找失败时，其返回的迭代器指向的是容器中最后一个键值对之后的位置，即不指向任何有意义的键值对，也就没有所谓的 first 和 second 成员了。

如果以上方法都不适用，我们还可以遍历整个 map 容器，找到包含指定键的键值对，进而获取该键对应的值。比如：

```
#include <iostream>
#include <map>      // map
#include <string>   // string
using namespace std;


int main() {
    //创建并初始化 map 容器
    std::map<std::string, std::string>myMap{ {"STL教程","http://c.biancheng.net/stl/"},
                                             {"C语言教程","http://c.biancheng.net/c/"},
                                             {"Java教程","http://c.biancheng.net/java/"} };


    for (auto iter = myMap.begin(); iter != myMap.end(); ++iter) {
        //调用 string 类的 compare() 方法，找到一个键和指定字符串相同的键值对
        if (!iter->first.compare("C语言教程")) {
            cout << iter->first << " " << iter->second << endl;
        }
    }
    return 0;
}
```
C语言教程 http://c.biancheng.net/c/

（3）map容器insert()插入数据的4种方式

Ⅰ.无需指定插入位置，直接将键值对添加到map容器中。insert()方法的语法格式有以下两种：

```
//1、引用传递一个键值对
pair<iterator,bool> insert (const value_type& val);
//2、以右值引用的方式传递键值对
template <class P>
    pair<iterator,bool> insert (P&& val);
```

其中，val 参数表示键值对变量，同时该方法会返回一个 pair 对象，其中 pair.first 表示一个迭代器，pair.second 为一个 bool 类型变量：

- 如果成功插入 val，则该迭代器指向新插入的 val，bool 值为 true；
- 如果插入 val 失败，则表明当前 map 容器中存有和 val 的键相同的键值对（用 p 表示），此时返回的迭代器指向 p，bool 值为 false。

```
#include <iostream>
#include <map>  //map
#include <string> //string
using namespace std;
int main()
{
    //创建一个空 map 容器
    std::map<string, string> mymap;

    //创建一个真实存在的键值对变量
    std::pair<string, string> STL = { "STL教程","http://c.biancheng.net/stl/" };

    //创建一个接收 insert() 方法返回值的 pair 对象
    std::pair<std::map<string, string>::iterator, bool> ret;

    //插入 STL，由于 STL 并不是临时变量，因此会以第一种方式传参
    ret = mymap.insert(STL);
    cout << "ret.iter = <{" << ret.first->first << ", " << ret.first->second << "}, " << ret.second << ">" << endl;
    //以右值引用的方式传递临时的键值对变量
    ret = mymap.insert({ "C语言教程","http://c.biancheng.net/c/" });
    cout << "ret.iter = <{" << ret.first->first << ", " << ret.first->second << "}, " << ret.second << ">" << endl;
    //插入失败样例
    ret = mymap.insert({ "STL教程","http://c.biancheng.net/java/" });
    cout << "ret.iter = <{" << ret.first->first << ", " << ret.first->second << "}, " << ret.second << ">" << endl;
```

```
    return 0;
}
ret.iter = <{STL教程, http://c.biancheng.net/stl/}, 1>
ret.iter = <{C语言教程, http://c.biancheng.net/c/}, 1>
ret.iter = <{STL教程, http://c.biancheng.net/stl/}, 0>
```

从执行结果中不难看出，程序中共执行了 3 次插入操作，其中成功了 2 次，失败了 1 次：

- 对于插入成功的 insert() 方法，其返回的 pair 对象中包含一个指向新插入键值对的迭代器和值为 1 的 bool 变量；
- 对于插入失败的 insert() 方法，同样会返回一个 pair 对象，其中包含一个指向 map 容器中键为 "STL教程" 的键值对和值为 0 的 bool 变量。

另外，在程序中的第 21 行代码，还可以使用如下 2 种方式创建临时的键值对变量，它们是等价的：

```
//调用 pair 类模板的构造函数
ret = mymap.insert(pair<string,string>{ "C语言教程","http://c.biancheng.net/c/" });
//调用 make_pair() 函数
ret = mymap.insert(make_pair("C语言教程", "http://c.biancheng.net/c/"));
```

Ⅱ. insert() 方法还支持向 map 容器的指定位置插入新键值对，该方法的语法格式如下：

```
//以普通引用的方式传递 val 参数
iterator insert (const_iterator position, const value_type& val);
//以右值引用的方式传递 val 键值对参数
template <class P>
    iterator insert (const_iterator position, P&& val);
```

其中 val 为要插入的键值对变量。注意，和第 1 种方式的语法格式不同，这里 insert() 方法返回的是迭代器，而不再是 pair 对象：

- 如果插入成功，insert() 方法会返回一个指向 map 容器中已插入键值对的迭代器；
- 如果插入失败，insert() 方法同样会返回一个迭代器，该迭代器指向 map 容器中和 val 具有相同键的那个键值对。

```
#include <iostream>
#include <map>  //map
#include <string> //string
using namespace std;
int main()
{
    //创建一个空 map 容器
    std::map<string, string> mymap;

    //创建一个真实存在的键值对变量
    std::pair<string, string> STL = { "STL教程","http://c.biancheng.net/stl/" };
    //指定要插入的位置
    std::map<string, string>::iterator it = mymap.begin();
    //向 it 位置以普通引用的方式插入 STL
    auto iter1 = mymap.insert(it, STL);
    cout << iter1->first << " " << iter1->second << endl;
    //向 it 位置以右值引用的方式插入临时键值对
    auto iter2 = mymap.insert(it, std::pair<string, string>("C语言教程", "http://c.biancheng.net/c/"));
    cout << iter2->first << " " << iter2->second << endl;
    //插入失败样例
    auto iter3 = mymap.insert(it, std::pair<string, string>("STL教程", "http://c.biancheng.net/java/"));
    cout << iter3->first << " " << iter3->second << endl;
    return 0;
}
STL教程 http://c.biancheng.net/stl/
C语言教程 http://c.biancheng.net/c/
STL教程 http://c.biancheng.net/stl/
```

再次强调，即便指定了新键值对的插入位置，map 容器仍会对存储的键值对进行排序。也可以说，决定新插入键值对位于 map 容器中位置的，不是 insert() 方法中传入的迭代器，而是新键值对中键的值。

Ⅲ. insert() 方法还支持向当前 map 容器中插入其它 map 容器指定区域内的所有键值对，该方法的语法格式如下：

```
template <class InputIterator>
  void insert (InputIterator first, InputIterator last);
```

其中 first 和 last 都是迭代器，它们的组合<first，last>可以表示某 map 容器中的指定区域。

```
#include <iostream>
#include <map>  //map
#include <string> //string
using namespace std;
int main()
{
    //创建并初始化 map 容器
    std::map<std::string, std::string>mymap{ {"STL教程","http://c.biancheng.net/stl/"},
                                             {"C语言教程","http://c.biancheng.net/c/"},
                                             {"Java教程","http://c.biancheng.net/java/"} };
    //创建一个空 map 容器
    std::map<std::string, std::string>copymap;
    //指定插入区域
    std::map<string, string>::iterator first = ++mymap.begin();
    std::map<string, string>::iterator last = mymap.end();
    //将<first,last>区域内的键值对插入到 copymap 中
    copymap.insert(first, last);
    //遍历输出 copymap 容器中的键值对
    for (auto iter = copymap.begin(); iter != copymap.end(); ++iter) {
        cout << iter->first << " " << iter->second << endl;
    }
    return 0;
}
Java教程 http://c.biancheng.net/java/
STL教程 http://c.biancheng.net/stl/
```

IV. insert() 方法还允许一次向 map 容器中插入多个键值对，其语法格式为：

```
void insert ({val1, val2, ...});
```

其中，val_i 都表示的是键值对变量。

```
#include <iostream>
#include <map>  //map
#include <string> //string
using namespace std;
int main()
{
    //创建空的 map 容器
    std::map<std::string, std::string>mymap;
    //向 mymap 容器中添加 3 个键值对
    mymap.insert({ {"STL教程", "http://c.biancheng.net/stl/"},
                   { "C语言教程","http://c.biancheng.net/c/" },
                   { "Java教程","http://c.biancheng.net/java/" } });


    for (auto iter = mymap.begin(); iter != mymap.end(); ++iter) {
        cout << iter->first << " " << iter->second << endl;
    }
    return 0;
}
```

```
C语言教程 http://c.biancheng.net/c/
Java教程 http://c.biancheng.net/java/
STL教程 http://c.biancheng.net/stl/
```

（4）map容器emplace()/emplace_hint()

和 insert() 方法相比，emplace() 和 emplace_hint() 方法的使用要简单很多，因为它们各自只有一种语法格式。其中，emplace() 方法的语法格式如下：

```
template <class... Args>
  pair<iterator,bool> emplace (Args&&... args);
```

参数 (Args&&... args) 指的是，这里只需要将创建新键值对所需的数据作为参数直接传入即可，此方法可以自行利用这些数据构建出指定的键值对。另外，该方法的返回值也是一个 pair 对象，其中 pair.first 为一个迭代器，pair.second 为一个 bool 类型变量：

- 当该方法将键值对成功插入到 map 容器中时，其返回的迭代器指向该新插入的键值对，同时 bool 变量的值为 true；
- 当插入失败时，则表明 map 容器中存在具有相同键的键值对，此时返回的迭代器指向此具有相同键的键值对，同时 bool 变量的值为 false。

```
#include <iostream>
#include <map>  //map
#include <string> //string
using namespace std;


int main()
{
    //创建并初始化 map 容器
    std::map<string, string>mymap;
    //插入键值对
    pair<map<string, string>::iterator, bool> ret = mymap.emplace("STL教程", "http://c.biancheng.net/stl/");
    cout << "1、ret.iter = <{" << ret.first->first << ", " << ret.first->second << "}, " << ret.second << ">" << endl;
    //插入新键值对
    ret = mymap.emplace("C语言教程", "http://c.biancheng.net/c/");
    cout << "2、ret.iter = <{" << ret.first->first << ", " << ret.first->second << "}, " << ret.second << ">" << endl;


    //失败插入的样例
    ret = mymap.emplace("STL教程", "http://c.biancheng.net/java/");
    cout << "3、ret.iter = <{" << ret.first->first << ", " << ret.first->second << "}, " << ret.second << ">" << endl;
    return 0;
}
```

```
1、ret.iter = <{STL教程, http://c.biancheng.net/stl/}, 1>
2、ret.iter = <{C语言教程, http://c.biancheng.net/c/}, 1>
3、ret.iter = <{STL教程, http://c.biancheng.net/stl/}, 0>
```

可以看到，程序中共执行了 3 次向 map 容器插入键值对的操作，其中前 2 次都成功了，第 3 次由于要插入的键值对的键和 map 容器中已存在的键值对的键相同，因此插入失败。

emplace_hint() 方法的功能和 emplace() 类似，其语法格式如下：

```
template <class... Args>
  iterator emplace_hint (const_iterator position, Args&&... args);
```

显然和 emplace() 语法格式相比，有以下 2 点不同：

- 该方法不仅要传入创建键值对所需要的数据，还需要传入一个迭代器作为第一个参数，指明要插入的位置（新键值对键会插入到该迭代器指向的键值对的前面）；
- 该方法的返回值是一个迭代器，而不再是 pair 对象。当成功插入新键值对时，返回的迭代器指向新插入的键值对；反之，如果插入失败，则表明 map 容器中存有相同键的键值对，返回的迭代器就指向这个键值对。

```
#include <iostream>
#include <map>  //map
#include <string> //string
using namespace std;


int main()
{
```

```
    //创建并初始化 map 容器
    std::map<string, string>mymap;
    //指定在 map 容器插入键值对
    map<string, string>::iterator iter = mymap.emplace_hint(mymap.begin(),"STL教程", "http://c.biancheng.net/stl/");
    cout << iter->first << " " << iter->second << endl;

    iter = mymap.emplace_hint(mymap.begin(), "C语言教程", "http://c.biancheng.net/c/");
    cout << iter->first << " " << iter->second << endl;

    //插入失败样例
    iter = mymap.emplace_hint(mymap.begin(), "STL教程", "http://c.biancheng.net/java/");
    cout << iter->first << " " << iter->second << endl;
    return 0;
}
```
```
STL教程 http://c.biancheng.net/stl/
C语言教程 http://c.biancheng.net/c/
STL教程 http://c.biancheng.net/stl/
```

注意，和 insert() 方法一样，虽然 emplace_hint() 方法指定了插入键值对的位置，但 map 容器为了保持存储键值对的有序状态，可能会移动其位置。

9、multimap容器用法总结

9.1 multimap的简单介绍

multimap 容器具有和 map 相同的特性，即 multimap 容器也用于存储 pair<const K, T> 类型的键值对（其中 K 表示键的类型，T 表示值的类型），其中各个键值对的键的值不能做修改；并且，该容器也会自行根据键的大小对存储的所有键值对做排序操作。和 map 容器的区别在于，multimap 容器中可以同时存储多（≥2）个键相同的键值对。

9.2 multimap的函数使用

9.2.1创建multimap容器

multimap 类模板内部提供有多个构造函数，总的来说，创建 multimap 容器的方式可归为以下 5 种。

（1）通过调用multimap类模板的默认构造函数，可以创建一个空的multimap容器

```
std::multimap<std::string, std::string>mymultimap;
```

（2）初始化multimap容器

```
//创建并初始化 multimap 容器
multimap<string, string>mymultimap{ {"C语言教程", "http://c.biancheng.net/c/"},
                                    {"Python教程", "http://c.biancheng.net/python/"},
                                    {"STL教程", "http://c.biancheng.net/stl/"} };
```

注意，使用此方式初始化 multimap 容器时，其底层会先将每一个{key, value}创建成 pair 类型的键值对，然后再用已建好的各个键值对初始化 multimap 容器。 实际上，我们完全可以先手动创建好键值对，然后再用其初始化 multimap 容器。下面程序使用了 2 种方式创建 pair 类型键值对，再用其初始化 multimap 容器，它们是完全等价的：

```
//借助 pair 类模板的构造函数来生成各个pair类型的键值对
multimap<string, string>mymultimap{
    pair<string,string>{"C语言教程", "http://c.biancheng.net/c/"},
    pair<string,string>{ "Python教程", "http://c.biancheng.net/python/"},
    pair<string,string>{ "STL教程", "http://c.biancheng.net/stl/"}
};
//调用 make_pair() 函数，生成键值对元素
//创建并初始化 multimap 容器
multimap<string, string>mymultimap{
    make_pair("C语言教程", "http://c.biancheng.net/c/"),
    make_pair("Python教程", "http://c.biancheng.net/python/"),
    make_pair("STL教程", "http://c.biancheng.net/stl/")
};
```

（3）通过调用 multimap 类模板的拷贝（复制）构造函数，也可以初始化新的 multimap 容器。

```
multimap<string, string>newmultimap(mymultimap);
```

在 C++ 11 标准中，还为 multimap 类增添了移动构造函数。即当有临时的 multimap 容器作为参数初始化新 multimap 容器时，其底层就会调用移动构造函数来实现初始化操作。

```
//创建一个会返回临时 multimap 对象的函数
multimap<string, string> dismultimap() {
    multimap<string, string>tempmultimap{ {"C语言教程", "http://c.biancheng.net/c/"},{"Python教程", "http://c.biancheng.net/python/"} };
    return tempmultimap;
}
//调用 multimap 类模板的移动构造函数创建 newMultimap 容器
multimap<string, string>newmultimap(dismultimap());
```

上面程序中，由于 dismultimap() 函数返回的 tempmultimap 容器是一个临时对象，因此在实现初始化 newmultimap 容器时，底层调用的是 multimap 容器的移动构造函数，而不再是拷贝构造函数。

（4）multimap 类模板还支持从已有 multimap 容器中，选定某块区域内的所有键值对，用作初始化新 multimap 容器时使用。

```
//创建并初始化 multimap 容器
multimap<string, string>mymultimap{ {"C语言教程", "http://c.biancheng.net/c/"},
                                    {"Python教程", "http://c.biancheng.net/python/"},
                                    {"STL教程", "http://c.biancheng.net/stl/"} };
multimap<string, string>newmultimap(++mymultimap.begin(), mymultimap.end());
```

（5）multimap 类模板共可以接收 4 个参数，其中第 3 个参数可用来修改 multimap 容器内部的排序规则。默认情况下，此参数的值为std::less<T>，这意味着以下 2 种创建 multimap 容器的方式是等价的：

```
multimap<char, int>mymultimap{ {'a',1},{'b',2} };
multimap<char, int, std::less<char>>mymultimap{ {'a',1},{'b',2} };
```

9.2.2 multimap容器的成员函数

成员函数	功能
begin()	返回指向容器中第一个（注意，是已排好序的第一个）键值对的双向迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
end()	返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
rbegin()	返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。
rend()	返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
find(key)	在 multimap 容器中查找键为 key 的键值对，如果成功找到，则返回指向该键值对的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
lower_bound(key)	返回一个指向当前multimap 容器中第一个大于或等于 key 的键值对的双向迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
upper_bound(key)	返回一个指向当前 multimap 容器中第一个大于 key 的键值对的迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
equal_range(key)	该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower_bound() 方法的返回值等价，pair.second 和 upper_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的键为 key 的键值对。
empty()	若容器为空，则返回 true；否则 false。
size()	返回当前 multimap 容器中存有键值对的个数。
max_size()	返回 multimap 容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。
insert()	向 multimap 容器中插入键值对。
erase()	删除 multimap 容器指定位置、指定键（key）值或者指定区域内的键值对。
swap()	交换 2 个 multimap 容器中存储的键值对，这意味着，操作的 2 个键值对的类型必须相同。
clear()	清空 multimap 容器中所有的键值对，即使 multimap 容器的 size() 为 0。
emplace()	在当前 multimap 容器中的指定位置处构造新键值对。其效果和插入键值对一样，但效率更高。
emplace_hint()	在本质上和 emplace() 在 multimap 容器中构造新键值对的方式是一样的，不同之处在于，使用者必须为该方法提供一个指示键值对生成位置的迭代器，并作为该方法的第一个参数。
count(key)	在当前 multimap 容器中，查找键为 key 的键值对的个数并返回。

和 map 容器相比，multimap 未提供 at() 成员方法，也没有重载 [] 运算符。这意味着，map 容器中通过指定键获取指定键值对的方式，将不再适用于 multimap 容器。其实这很好理解，因为 multimap 容器中指定的键可能对应多个键值对，而不再是 1 个。

10、set容器用法总结

10.1 set的简单介绍

C++STL标准库中还提供有 set 和 multiset 这 2 个容器，它们也属于关联式容器。 和 map、multimap 容器不同，使用 set 容器存储的各个键值对，要求键 key 和值 value 必须相等。

```
{<'a', 1>, <'b', 2>, <'c', 3>}
{<'a', 'a'>, <'b', 'b'>, <'c', 'c'>}
```

第一组数据中各键值对的键和值不相等，而第二组中各键值对的键和值对应相等。对于 set 容器来说，只能存储第 2 组键值对，而无法存储第一组键值对。 基于 set 容器的这种特性，当使用 set 容器存储键值对时，只需要为其提供各键值对中的 value 值（也就是 key 的值）即可。仍以存储上面第 2 组键值对为例，只需要为 set 容器提供 {'a','b','c'} ，该容器即可成功将它们存储起来。set容器能够自动根据key排序，也就等价为根据value排序。 另外，使用 set 容器存储的各个元素的值必须各不相同。更重要的是，从语法上讲 set 容器并没有强制对存储元素的类型做 const 修饰，即 set 容器中存储的元素的值是可以修改的。但是，C++ 标准为了防止用户修改容器中元素的值，对所有可能会实现此操作的行为做了限制，使得在正常情况下，用户是无法做到修改 set 容器中元素的值的。

10.2 set的函数使用

10.2.1创建set容器

（1）调用默认构造函数，创建空的set容器

```
std::set<std::string> myset;
```

（2）创建时初始化set容器

```
std::set<std::string> myset{"http://c.biancheng.net/java/",
                            "http://c.biancheng.net/stl/",
                            "http://c.biancheng.net/python/"};
```

由此即创建好了包含 3 个 string 元素的 myset 容器。由于其采用默认的 std::less<T> 规则，因此其内部存储 string 元素的顺序如下所示：

```
"http://c.biancheng.net/java/"
"http://c.biancheng.net/python/"
"http://c.biancheng.net/stl/"
```

（3）set 类模板中还提供了拷贝（复制）构造函数，可以实现在创建新 set 容器的同时，将已有 set 容器中存储的所有元素全部复制到新 set 容器中。

```
std::set<std::string> copyset(myset);
//等同于
//std::set<std::string> copyset = myset
```

另外，C++ 11 标准还为 set 类模板新增了移动构造函数，其功能是实现创建新 set 容器的同时，利用临时的 set 容器为其初始化。

```
set<string> retSet() {
    std::set<std::string> myset{ "http://c.biancheng.net/java/",
                            "http://c.biancheng.net/stl/",
                            "http://c.biancheng.net/python/" };
    return myset;
}
std::set<std::string> copyset(retSet());
//或者
//std::set<std::string> copyset = retSet();
```

注意，由于 retSet() 函数的返回值是一个临时 set 容器，因此在初始化 copyset 容器时，其内部调用的是 set 类模板中的移动构造函数，而非拷贝构造函数。

（4）set 类模板还支持取已有 set 容器中的部分元素，来初始化新 set 容器。

```
std::set<std::string> myset{ "http://c.biancheng.net/java/",
                    "http://c.biancheng.net/stl/",
                    "http://c.biancheng.net/python/" };
std::set<std::string> copyset(++myset.begin(), myset.end());
```

（5）以上几种方式创建的 set 容器，都采用了默认的std::less<T>规则。其实，借助 set 类模板定义中第 2 个参数，我们完全可以手动修改 set 容器中的排序规则。

```
std::set<std::string,std::greater<string> > myset{
    "http://c.biancheng.net/java/",
    "http://c.biancheng.net/stl/",
    "http://c.biancheng.net/python/"};
"http://c.biancheng.net/stl/"
"http://c.biancheng.net/python/"
"http://c.biancheng.net/java/"
```

10.2.2 set容器的成员函数

成员函数	功能
begin()	返回指向容器中第一个（注意，是已排好序的第一个）键值对的双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
end()	返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。

	begin() 结合使用。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
rbegin()	返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。
rend()	返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
find(val)	在 set 容器中查找值为 val 的元素，如果成功找到，则返回指向该元素的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
lower_bound(val)	返回一个指向当前 set 容器中第一个大于或等于 val 的元素的双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
upper_bound(val)	返回一个指向当前 set 容器中第一个大于 val 的元素的迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
equal_range(val)	该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower_bound() 方法的返回值等价，pair.second 和 upper_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的值为 val 的元素（set 容器中各个元素是唯一的，因此该范围最多包含一个元素）。
empty()	若容器为空，则返回 true；否则 false。
size()	返回当前 set 容器中存有元素的个数。
max_size()	返回 set 容器所能容纳元素的最大个数，不同的操作系统，其返回值亦不相同。
insert()	向 set 容器中插入元素。
erase()	删除 set 容器中存放的元素。
swap()	交换 2 个 set 容器中存储的所有元素。这意味着，操作的 2 个 set 容器的类型必须相同。
clear()	清空 set 容器中所有的元素，即使 set 容器的 size() 为 0。
emplace()	在当前 set 容器中的指定位置处构造新元素。其效果和insert()一样，但效率更高。
emplace_hint()	在本质上和 emplace() 在 set 容器中构造新元素的方式是一样的，不同之处在于，使用者必须为该方法提供一个指示新元素生成位置的迭代器，并作为该方法的第一个参数。
count(val)	在当前 set 容器中，查找值为 val 的元素的个数，并返回。注意，由于 set 容器中各元素的值是唯一的，因此该函数的返回值最大为 1。

（1）set容器迭代器用法

和 map 容器不同，C++STL中的 set 容器类模板中未提供 at() 成员函数，也未对 [] 运算符进行重载。因此，要想访问 set 容器中存储的元素，只能借助 set 容器的迭代器。 值得一提的是，C++ STL 标准库为 set 容器配置的迭代器类型为双向迭代器。这意味着，假设 p 为此类型的迭代器，则其只能进行 ++p、p++、--p、p--、*p 操作，并且 2 个双向迭代器之间做比较，也只能使用 == 或者 != 运算符。

Ⅰ.begin()/end()

```
#include <iostream>
#include <set>
#include <string>
using namespace std;


int main()
{
    //创建并初始化set容器
    std::set<std::string> myset{ "http://c.biancheng.net/java/",
                                 "http://c.biancheng.net/stl/",
```

```
                                "http://c.biancheng.net/python/"
    };
    //利用双向迭代器，遍历myset
    for (auto iter = myset.begin(); iter != myset.end(); ++iter) {
        cout << *iter << endl;
    }
    return 0;
}
```

Ⅱ.find()

除此之外，如果只想遍历 set 容器中指定区域内的部分数据，则可以借助 find()、lower_bound() 以及 upper_bound() 实现。通过调用它们，可以获取一个指向指定元素的迭代器。 需要特别指出的是，equal_range(val) 函数的返回值是一个 pair 类型数据，其包含 2 个迭代器，表示 set 容器中和指定参数 val 相等的元素所在的区域，但由于 set 容器中存储的元素各不相等，因此该函数返回的这 2 个迭代器所表示的范围中，最多只会包含 1 个元素。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;


int main()
{
    //创建并初始化set容器
    std::set<std::string> myset{ "http://c.biancheng.net/java/",
                                 "http://c.biancheng.net/stl/",
                                 "http://c.biancheng.net/python/"
    };

    set<string>::iterator iter = myset.find("http://c.biancheng.net/python/");
    for (;iter != myset.end();++iter)
    {
        cout << *iter << endl;
    }
    return 0;
}
```

```
http://c.biancheng.net/python/
http://c.biancheng.net/stl/
```

（2）insert()的四种用法

Ⅰ.只要给定目标元素的值，insert()方法即可将该元素添加到set容器中，其语法格式如下：

```
//普通引用方式传参
pair<iterator,bool> insert (const value_type& val);
//右值引用方式传参
pair<iterator,bool> insert (value_type&& val);
```

可以看到，以上 2 种语法格式的 insert() 方法，返回的都是 pair 类型的值，其包含 2 个数据，一个迭代器和一个 bool 值：

- 当向 set 容器添加元素成功时，该迭代器指向 set 容器新添加的元素，bool 类型的值为 true；
- 如果添加失败，即证明原 set 容器中已存有相同的元素，此时返回的迭代器就指向容器中相同的此元素，同时 bool 类型的值为 false。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;


int main()
{
    //创建并初始化set容器
    std::set<std::string> myset;
    //准备接受 insert() 的返回值
    pair<set<string>::iterator, bool> retpair;
    //采用普通引用传值方式
    string str = "http://c.biancheng.net/stl/";
    retpair = myset.insert(str);
    cout << "iter->" << *(retpair.first) << " " << "bool = " << retpair.second << endl;
    //采用右值引用传值方式
    retpair = myset.insert("http://c.biancheng.net/python/");
    cout << "iter->" << *(retpair.first) << " " << "bool = " << retpair.second << endl;
    return 0;
}
```

```
iter->http://c.biancheng.net/stl/ bool = 1
iter->http://c.biancheng.net/python/ bool = 1
```

Ⅱ.insert()还可以指定将新元素插入到set容器中的具体位置，其语法格式如下：

```
//以普通引用的方式传递 val 值
iterator insert (const_iterator position, const value_type& val);
//以右值引用的方式传递 val 值
iterator insert (const_iterator position, value_type&& val);
```

以上 2 种语法格式中，insert() 函数的返回值为迭代器：

- 当向 set 容器添加元素成功时，该迭代器指向容器中新添加的元素；
- 当添加失败时，证明原 set 容器中已有相同的元素，该迭代器就指向 set 容器中相同的这个元素。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
```

```
int main()
{
    //创建并初始化set容器
    std::set<std::string> myset;
    //准备接受 insert() 的返回值
    set<string>::iterator iter;

    //采用普通引用传值方式
    string str = "http://c.biancheng.net/stl/";
    iter = myset.insert(myset.begin(),str);
    cout << "myset size =" << myset.size() << endl;

    //采用右值引用传值方式
    iter = myset.insert(myset.end(),"http://c.biancheng.net/python/");
    cout << "myset size =" << myset.size() << endl;
    return 0;
}
myset size =1
myset size =2
```

注意，使用 insert() 方法将目标元素插入到 set 容器指定位置后，如果该元素破坏了容器内部的有序状态，set 容器还会自行对新元素的位置做进一步调整。也就是说，insert() 方法中指定新元素插入的位置，并不一定就是该元素最终所处的位置。

Ⅲ.insert()方法支持向当前 set 容器中插入其它 set 容器指定区域内的所有元素，只要这 2 个 set 容器存储的元素类型相同即可。

insert() 方法的语法格式如下：

```
template <class InputIterator>
  void insert (InputIterator first, InputIterator last);
```

其中 first 和 last 都是迭代器，它们的组合 [first,last) 可以表示另一 set 容器中的一块区域，该区域包括 first 迭代器指向的元素，但不包含 last 迭代器指向的元素。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
{
    //创建并初始化set容器
    std::set<std::string> myset{ "http://c.biancheng.net/stl/",
                                 "http://c.biancheng.net/python/",
                                 "http://c.biancheng.net/java/" };
    //创建一个同类型的空 set 容器
    std::set<std::string> otherset;
    //利用 myset 初始化 otherset
    otherset.insert(++myset.begin(), myset.end());
    //输出 otherset 容器中的元素
    for (auto iter = otherset.begin(); iter != otherset.end(); ++iter) {
        cout << *iter << endl;
    }
    return 0;
}
http://c.biancheng.net/python/
http://c.biancheng.net/stl/
```

IV.可以一次向set容器中添加多个元素

```
void insert ( {E1, E2,...,En} );
```

其中，E_i 表示新添加的元素。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
{
    //创建并初始化set容器
    std::set<std::string> myset;
    //向 myset 中添加多个元素
    myset.insert({ "http://c.biancheng.net/stl/",
        "http://c.biancheng.net/python/",
        "http://c.biancheng.net/java/" });

    for (auto iter = myset.begin(); iter != myset.end(); ++iter) {
        cout << *iter << endl;
    }
    return 0;
}
http://c.biancheng.net/java/
http://c.biancheng.net/python/
http://c.biancheng.net/stl/
```

（3）emplace()和emplace_hint()

emplace() 和 emplace_hint() 是 C++ 11 标准加入到 set 类模板中的，相比具有同样功能的 insert() 方法，完成同样的任务，emplace() 和 emplace_hint() 的效率会更高。

emplace() 方法的语法格式如下：

```
template <class... Args>
  pair<iterator,bool> emplace (Args&&... args);
```

其中，参数 (Args&&... args) 指的是，只需要传入构建新元素所需的数据即可，该方法可以自行利用这些数据构建出要添加的元素。比如，若 set 容器中存储的元素类型为自定义的结构体或者类，则在使用 emplace() 方法向容器中添加新元素时，构造新结构体变量（或者类对象）需要多少个数据，就需要为该方法传入相应个数的数据。

另外，该方法的返回值类型为 pair 类型，其包含 2 个元素，一个迭代器和一个 bool 值：

- 当该方法将目标元素成功添加到 set 容器中时，其返回的迭代器指向新插入的元素，同时 bool 值为 true；
- 当添加失败时，则表明原 set 容器中已存在相同值的元素，此时返回的迭代器指向容器中具有相同键的这个元素，同时 bool 值为 false。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
int main()
{
    //创建并初始化 set 容器
    std::set<string>myset;
    //向 myset 容器中添加元素
    pair<set<string, string>::iterator, bool> ret = myset.emplace("http://c.biancheng.net/stl/");
    cout << "myset size = " << myset.size() << endl;
    cout << "ret.iter = <" << *(ret.first) << ", " << ret.second << ">" << endl;
    return 0;
}
myset size = 1
ret.iter = <http://c.biancheng.net/stl/, 1>
```

emplace_hint() 方法的功能和 emplace() 类似，其语法格式如下：

```
template <class... Args>
  iterator emplace_hint (const_iterator position, Args&&... args);
```

和 emplace() 方法相比，有以下 2 点不同：

- 该方法需要额外传入一个迭代器，用来指明新元素添加到 set 容器的具体位置（新元素会添加到该迭代器指向元素的前面）；
- 返回值是一个迭代器，而不再是 pair 对象。当成功添加元素时，返回的迭代器指向新添加的元素；反之，如果添加失败，则迭代器就指向 set 容器和要添加元素的值相同的元素。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
int main()
{
    //创建并初始化 set 容器
    std::set<string>myset;
    //在 set 容器的指定位置添加键值对
    set<string>::iterator iter = myset.emplace_hint(myset.begin(), "http://c.biancheng.net/stl/");
    cout << "myset size = " << myset.size() << endl;
    cout << *iter << endl;
    return 0;
}
myset size = 1
http://c.biancheng.net/stl/
```

11、multiset容器用法总结

11.1 multiset的简单介绍

set 容器具有以下几个特性：

- 不再以键值对的方式存储数据，因为 set 容器专门用于存储键和值相等的键值对，因此该容器中真正存储的是各个键值对的值（value）；
- set 容器在存储数据时，会根据各元素值的大小对存储的元素进行排序（默认做升序排序）；
- 存储到 set 容器中的元素，虽然其类型没有明确用 const 修饰，但正常情况下它们的值是无法被修改的；
- set 容器存储的元素必须互不相等。

在此基础上，C++STL标准库中还提供有一个和 set 容器相似的关联式容器，即 multiset 容器。所谓“相似”，是指 multiset 容器遵循 set 容器的前 3 个特性，仅在第 4 条特性上有差异。和 set 容器不同的是，multiset 容器可以存储多个值相同的元素。

11.2 multiset的函数使用

11.2.1创建multiset容器

创建 multiset 容器，无疑需要调用 multiset 类模板中的构造函数。值得一提的是，multiset 类模板提供的构造函数，和 set 类模板中提供创建 set 容器的构造函数，是完全相同的。这意味着，创建 set 容器的方式，也同样适用于创建 multiset

容器。

11.2.2 multiset容器的成员函数

成员函数	功能
begin()	返回指向容器中第一个（注意，是已排好序的第一个）键值对的双向迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
end()	返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
rbegin()	返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。
rend()	返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
crbegin()	和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
crend()	和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。
find(val)	在 multiset 容器中查找值为 val 的元素，如果成功找到，则返回指向该元素的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
lower_bound(val)	返回一个指向当前 multiset 容器中第一个大于或等于 val 的元素的双向迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
upper_bound(val)	返回一个指向当前 multiset 容器中第一个大于 val 的元素的迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。
equal_range(val)	该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower_bound() 方法的返回值等价，pair.second 和 upper_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的所有值为 val 的元素
empty()	若容器为空，则返回 true；否则 false。
size()	返回当前 multiset 容器中存有元素的个数。
max_size()	返回 multiset 容器所能容纳元素的最大个数，不同的操作系统，其返回值亦不相同。
insert()	向 multiset 容器中插入元素。
erase()	删除 multiset 容器中存放的指定元素。
swap()	交换 2 个 multiset 容器中存储的所有元素。这意味着，操作的 2 个 multiset 容器的类型必须相同。
clear()	清空 multiset 容器中所有的元素，即使 multiset 容器的 size() 为 0。
emplace()	在当前 multiset 容器中的指定位置处构造新元素。其效果和insert()一样，但效率更高。
emplace_hint()	在本质上和 emplace() 在 multiset 容器中构造新元素的方式是一样的，不同之处在于，使用者必须为该方法提供一个指示新元素生成位置的迭代器，并作为该方法的第一个参数。
count(val)	在当前 multiset 容器中，查找值为 val 的元素的个数，并返回。

```
#include <iostream>
#include <set>
#include <string>
using namespace std;


int main() {
    std::multiset<int> mymultiset{1,2,2,2,3,4,5};
    cout << "multiset size = " << mymultiset.size() << endl;
    cout << "multiset count(2) =" << mymultiset.count(2) << endl;
    //向容器中添加元素 8
    mymultiset.insert(8);
    //删除容器中所有值为 2 的元素
    int num = mymultiset.erase(2);
```

```
    cout << "删除了 " << num << " 个元素 2" << endl;
    //输出容器中存储的所有元素
    for (auto iter = mymultiset.begin(); iter != mymultiset.end(); ++iter) {
        cout << *iter << " ";
    }
    return 0;
}
multiset size = 7
multiset count(2) =3
删除了 3 个元素 2
1 3 4 5 8
```

12、无序关联式容器用法总结

12.1 简单介绍

除了序列式容器和关联式容器之外，C++ 11 标准库又引入了一类容器，即无序关联式容器。 无序关联式容器，又称哈希容器。和关联式容器一样，此类容器存储的也是键值对元素；不同之处在于，关联式容器默认情况下会对存储的元素做升序排序，而无序关联式容器不会。 和其它类容器相比，无序关联式容器擅长通过指定键查找对应的值，而遍历容器中存储元素的效率不如关联式容器。

和关联式容器一样，无序容器也使用键值对（pair 类型）的方式存储数据。不过，它们本质上是不同的：

- 关联式容器的底层实现采用的是树存储结构，更确切地说是红黑树结构；
- 无序容器的底层实现采用的是哈希表的存储结构。

C++STL底层采用哈希表实现无序容器时，会将所有数据存储到一整块连续的内存空间中，并且当数据存储位置发生冲突时，解决方法选用的是“链地址法”（又称“开链法”）。

基于底层实现采用了不同的数据结构，因此和关联式容器相比，无序容器具有以下2个特点：

- 无序容器内部存储的键值对是无序的，各键值对的存储位置取决于该键值对中的键；
- 和关联式容器相比，无序容器擅长通过指定键查找对应的值（平均时间复杂度为O(1)）；但对于使用迭代器遍历容器中存储的元素，无序容器的执行效率则不如关联式容器。

12.2 无序容器种类

和关联式容器一样，无序容器知识一类容器的统称，其包含有4各具体容器，分别为unordered_map、unordered_multimap、unordered_set以及unordered_multiset。

无序容器	功能
unordered_map	存储键值对<key，value>类型的元素，其中各个键值对键的值不允许重复，且该容器中存储的键值对是无序的。
unordered_multimap	和unordered_map唯一的区别在于，该容器允许存储多个键相同的键值对。
unordered_set	不再以键值对的形式存储数据，而是直接存储数元素本身（当然也可以理解为，该容器存储的全部都是键key和值value相等的键值对，正因为它们相等，因此只存储value即可）。另外，该容器存储的元素不能重复，且容器内部存储的元素也是无序的。
unordered_multiset	和unordered_set唯一的区别在于，该容器允许存储值相同的元素。

总的来说，实际场景中如果涉及大量遍历容器的操作，建议首选关联式容器；反之，如果更多的操作是通过键获取对应的值，则应首选无序容器。

12.3 unordered_map容器用法

12.3.1 容器说明

unordered_map 容器，直译过来就是"无序 map 容器"的意思。所谓“无序”，指的是 unordered_map 容器不会像 map 容器那样对存储的数据进行排序。换句话说，unordered_map 容器和 map 容器仅有一点不同，即 map 容器中存储的数据是有序的，而 unordered_map 容器中是无序的。

具体来讲，unordered_map 容器和 map 容器一样，以键值对（pair类型）的形式存储数据，存储的各个键值对的键互不相同且不允许被修改。但由于 unordered_map 容器底层采用的是哈希表存储结构，该结构本身不具有对数据的排序功能，所以此容器内部不会自行对存储的键值对进行排序。

unordered_map 容器模板的定义如下所示：

```
template < class Key,                          //键值对中键的类型
           class T,                            //键值对中值的类型
           class Hash = hash<Key>,             //容器内部存储键值对所用的哈希函数
           class Pred = equal_to<Key>,         //判断各个键值对键相同的规则
           class Alloc = allocator< pair<const Key,T> >  // 指定分配器对象的类型
           > class unordered_map;
```

参数	含义
<key，T>	前两个参数分别用于确定键值对中键和值的类型，也就是存储键值对的类型。
Hash=hash<Key>	用于指明容器在存储各个键值对时要使用的哈希函数，默认使用 STL 标准库提供的 hash<key> 哈希函数。注意，默认哈希函数只适用于基本数据类型（包括 string 类型），而不适用于自定义的结构体或者类。
Pred = equal_to<Key>	要知道，unordered_map 容器中存储的各个键值对的键是不能相等的，而判断是否相等的规则，就由此参数指定。默认情况下，使用 STL 标准库中提供的 equal_to<key> 规则，该规则仅支持可直接用 == 运算符做比较的数据类型。

12.3.2 创建容器的方法

（1）调用unordered_map模板类的默认构造函数，可以创建空的unordered_map容器。

```
std::unordered_map<std::string, std::string> umap;
```

（2）初始化容器。

```
std::unordered_map<std::string, std::string> umap{
    {"Python教程","http://c.biancheng.net/python/"},
    {"Java教程","http://c.biancheng.net/java/"},
    {"Linux教程","http://c.biancheng.net/linux/"} };
```

（3）可以调用 unordered_map 模板中提供的复制（拷贝）构造函数，将现有 unordered_map 容器中存储的键值对，复制给新建 unordered_map 容器。

```
std::unordered_map<std::string, std::string> umap2(umap);
```

除此之外，C++ 11 标准中还向 unordered_map 模板类增加了移动构造函数，即以右值引用的方式将临时 unordered_map 容器中存储的所有键值对，全部复制给新建容器。例如：

```
//返回临时 unordered_map 容器的函数
std::unordered_map <std::string, std::string > retUmap(){
    std::unordered_map<std::string, std::string>tempUmap{
        {"Python教程","http://c.biancheng.net/python/"},
        {"Java教程","http://c.biancheng.net/java/"},
        {"Linux教程","http://c.biancheng.net/linux/"} };
    return tempUmap;
}
//调用移动构造函数，创建 umap2 容器
std::unordered_map<std::string, std::string> umap2(retUmap());
```

（4）如果不想全部拷贝，可以使用 unordered_map 类模板提供的迭代器，在现有 unordered_map 容器中选择部分区域内的键值对，为新建 unordered_map 容器初始化。

```
//传入 2 个迭代器，
std::unordered_map<std::string, std::string> umap2(++umap.begin(),umap.end());
```

12.3.3 unordered_map容器的成员方法

成员方法	功能
begin()	返回指向容器中第一个键值对的正向迭代器。
end()	返回指向容器中最后一个键值对之后位置的正向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。
empty()	若容器为空，则返回 true；否则 false。
size()	返回当前容器中存有键值对的个数。
max_size()	返回容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。

operator[key]	该模板类中重载了 [] 运算符，其功能是可以向访问数组中元素那样，只要给定某个键值对的键 key，就可以获取该键对应的值。注意，如果当前容器中没有以 key 为键的键值对，则其会使用该键向当前容器中插入一个新键值对。
at(key)	返回容器中存储的键 key 对应的值，如果 key 不存在，则会抛出 out_of_range 异常。
find(key)	查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如果 end() 方法返回的迭代器）。
count(key)	在容器中查找以 key 键的键值对的个数。
equal_range(key)	返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中键为 key 的键值对所在的范围。
emplace()	向容器中添加新键值对，效率比 insert() 方法高。
emplace_hint()	向容器中添加新键值对，效率比 insert() 方法高。
insert()	向容器中添加新键值对。
erase()	删除指定键值对。
clear()	清空容器，即删除容器中存储的所有键值对。
swap()	交换 2 个 unordered_map 容器存储的键值对，前提是必须保证这 2 个容器的类型完全相等。
bucket_count()	返回当前容器底层存储键值对时，使用桶（一个线性链表代表一个桶）的数量。
max_bucket_count()	返回当前系统中，unordered_map 容器底层最多可以使用多少桶。
bucket_size(n)	返回第 n 个桶中存储键值对的数量。
bucket(key)	返回以 key 为键的键值对所在桶的编号。
load_factor()	返回 unordered_map 容器中当前的负载因子。负载因子，指的是的当前容器中存储键值对的数量（size()）和使用桶数（bucket_count()）的比值，即 load_factor() = size() / bucket_count()。
max_load_factor()	返回或者设置当前 unordered_map 容器的负载因子。
rehash(n)	将当前容器底层使用桶的数量设置为 n。
reserve()	将存储桶的数量（也就是 bucket_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。
hash_function()	返回当前容器使用的哈希函数对象。

（1）unordered_map迭代器用法

C++ STL 标准库中，unordered_map 容器迭代器的类型为前向迭代器（又称正向迭代器）。这意味着，假设 p 是一个前向迭代器，则其只能进行 *p、p++、++p 操作，且 2 个前向迭代器之间只能用 == 和 != 运算符做比较。

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    //创建 umap 容器
    unordered_map<string, string> umap{
        {"Python教程","http://c.biancheng.net/python/"},
        {"Java教程","http://c.biancheng.net/java/"},
        {"Linux教程","http://c.biancheng.net/linux/"} };


    cout << "umap 存储的键值对包括：" << endl;
    //遍历输出 umap 容器中所有的键值对
    for (auto iter = umap.begin(); iter != umap.end(); ++iter) {
        cout << "<" << iter->first << ", " << iter->second << ">" << endl;
    }
    //获取指向指定键值对的前向迭代器
```

```
    unordered_map<string, string>::iterator iter = umap.find("Java教程");
    cout <<"umap.find(\"Java教程\") = " << "<" << iter->first << ", " << iter->second << ">" << endl;
    return 0;
}
umap 存储的键值对包括：
<Python教程, http://c.biancheng.net/python/>
<Linux教程, http://c.biancheng.net/linux/>
<Java教程, http://c.biancheng.net/java/>
umap.find("Java教程") = <Java教程, http://c.biancheng.net/java/>
```

需要注意的是，在操作 unordered_map 容器过程（尤其是向容器中添加新键值对）中，一旦当前容器的负载因子超过最大负载因子（默认值为 1.0），该容器就会适当增加桶的数量（通常是翻一倍），并自动执行 rehash() 成员方法，重新调整各个键值对的存储位置（此过程又称“重哈希”），此过程很可能导致之前创建的迭代器失效。所谓迭代器失效，针对的是那些用于表示容器内某个范围地迭代器，由于重哈希会重新调整每个键值对的存储位置，所以容器重哈希之后，之前表示特定范围的迭代器很可能无法再正确表示该范围。但是，重哈希并不会影响那些指向单个键值对元素的迭代器。

（2）unordered_map获取元素的四种方法

Ⅰ.unordered_map 容器类模板中，实现了对 [ ] 运算符的重载，使得我们可以像“利用下标访问普通数组中元素”那样，通过目标键值对的键获取到该键对应的值。

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    //创建 umap 容器
    unordered_map<string, string> umap{
        {"Python教程","http://c.biancheng.net/python/"},
        {"Java教程","http://c.biancheng.net/java/"},
        {"Linux教程","http://c.biancheng.net/linux/"} };
    //获取 "Java教程" 对应的值
    string str = umap["Java教程"];
    cout << str << endl;
    return 0;
}
http://c.biancheng.net/java/
```

需要注意的是，如果当前容器中并没有存储以 [ ] 运算符内指定的元素作为键的键值对，则此时 [ ] 运算符的功能将转变为：向当前容器中添加以目标元素为键的键值对。举个例子：

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    //创建空 umap 容器
    unordered_map<string, string> umap;
    //[] 运算符在 = 右侧
    string str = umap["STL教程"];
    //[] 运算符在 = 左侧
    umap["C教程"] = "http://c.biancheng.net/c/";

    for (auto iter = umap.begin(); iter != umap.end(); ++iter) {
        cout << iter->first << " " << iter->second << endl;
    }
    return 0;
}
C教程 http://c.biancheng.net/c/
STL教程
```

可以看到，当使用 [ ] 运算符向 unordered_map 容器中添加键值对时，分为 2 种情况：

- 当 [ ] 运算符位于赋值号（=）右侧时，则新添加键值对的键为 [ ] 运算符内的元素，其值为键值对要求的值类型的默认值（string 类型默认值为空字符串）；
- 当 [ ] 运算符位于赋值号（=）左侧时，则新添加键值对的键为 [ ] 运算符内的元素，其值为赋值号右侧的元素。

Ⅱ.unordered_map 类模板中，还提供有 at() 成员方法，和使用 [ ] 运算符一样，at() 成员方法也需要根据指定的键，才能从容器中找到该键对应的值；不同之处在于，如果在当前容器中查找失败，该方法不会向容器中添加新的键值对，而是直接抛出out_of_range异常。

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    //创建 umap 容器
    unordered_map<string, string> umap{
        {"Python教程","http://c.biancheng.net/python/"},
        {"Java教程","http://c.biancheng.net/java/"},
        {"Linux教程","http://c.biancheng.net/linux/"} };
    //获取指定键对应的值
    string str = umap.at("Python教程");
    cout << str << endl;


    //执行此语句会抛出 out_of_range 异常
    //cout << umap.at("GO教程");
    return 0;
```

```
}
```
http://c.biancheng.net/python/

Ⅲ.[ ] 运算符和 at() 成员方法基本能满足大多数场景的需要。除此之外，还可以借助 unordered_map 模板中提供的 find() 成员方法。

和前面方法不同的是，通过 find() 方法得到的是一个正向迭代器，该迭代器的指向分以下 2 种情况：

- 当 find() 方法成功找到以指定元素作为键的键值对时，其返回的迭代器就指向该键值对；
- 当 find() 方法查找失败时，其返回的迭代器和 end() 方法返回的迭代器一样，指向容器中最后一个键值对之后的位置。

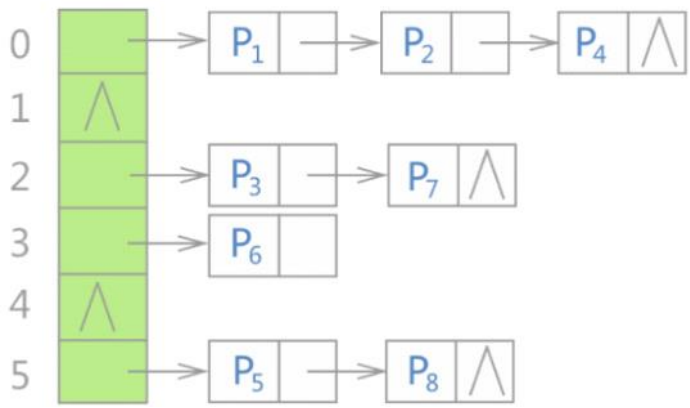
```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    //创建 umap 容器
    unordered_map<string, string> umap{
        {"Python教程","http://c.biancheng.net/python/"},
        {"Java教程","http://c.biancheng.net/java/"},
        {"Linux教程","http://c.biancheng.net/linux/"} };
    //查找成功
    unordered_map<string, string>::iterator iter = umap.find("Python教程");
    cout << iter->first << " " << iter->second << endl;
    //查找失败
    unordered_map<string, string>::iterator iter2 = umap.find("GO教程");
    if (iter2 == umap.end()) {
        cout << "当前容器中没有以\"GO教程\"为键的键值对";
    }
    return 0;
}
```
Python教程 http://c.biancheng.net/python/
当前容器中没有以"GO教程"为键的键值对

Ⅳ.除了 find() 成员方法之外，甚至可以借助 begin()/end() 或者 cbegin()/cend()，通过遍历整个容器中的键值对来找到目标键值对。

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    //创建 umap 容器
    unordered_map<string, string> umap{
        {"Python教程","http://c.biancheng.net/python/"},
        {"Java教程","http://c.biancheng.net/java/"},
        {"Linux教程","http://c.biancheng.net/linux/"} };
    //遍历整个容器中存储的键值对
    for (auto iter = umap.begin(); iter != umap.end(); ++iter) {
        //判断当前的键值对是否就是要找的
        if (!iter->first.compare("Java教程")) {
            cout << iter->second << endl;
            break;
        }
    }
    return 0;
}
```
http://c.biancheng.net/java/

12.3.4 无序容器的底层实现机制

C++ STL 标准库中，不仅是 unordered_map 容器，所有无序容器的底层实现都采用的是哈希表存储结构。更准确地说，是用“链地址法”（又称“开链法”）解决数据存储位置发生冲突的哈希表。



上图中Pi表示存储的各个键值对。可以看到，当使用无序容器存储键值对时，会先申请一整块连续的存储空间，但此空间并不用来直接存储键值对，而是存储各个链表的头指针，各键值对真正的存储位置是各个链表的节点。注意，STL标准库通常选用vector容器存储各个链表的头指针。

不仅如此，在C++STL标准库中，将上图中的各个链表称为桶（bucket），每个桶都有自己的编号（从0开始）。当有新键值对存储到无序容器中时，整个存储过程分为如下几步：

- 将该键值对中键的值带入设计好的哈希函数，会得到一个哈希值（一个整数，用H表示）；
- 将H和无序容器拥有桶的数量n做整除运算（即H%n），该结果即表示应将此键值对存储到的桶的编号；
- 建立一个新节点存储此键值对，同时将该节点链接到相应编号的桶上。

另外值得一提的是，哈希表存储结构还有一个重要的属性，称为负载因子（load factor）。该属性同样适用于无序容器，用于衡量容器存储键值对的空/满程序，即负载因子越大，意味着容器越满，即各链表中挂载着越多的键值对，这无疑会降低容器查找目标键值对的效率；反之，负载因子越小，容器肯定越空，但并不一定各个链表中挂载的键值对就越少。举个例子，如果设计的哈希函数不合理，使得各个键值对的键带入该函数得到的哈希值始终相同（所有键值对始终存储在同一链表上）。这种情况下，即便增加桶数是的负载因子减小，该容器的查找效率依旧很差。

无序容器中，负载因子的计算方法为：容器存储的总的键值对/桶数。

默认情况下，无序容器的最大负载因子为 1.0。如果操作无序容器过程中，使得最大复杂因子超过了默认值，则容器会自动增加桶数，并重新进行哈希，以此来减小负载因子的值。需要注意的是，此过程会导致容器迭代器失效，但指向单个键值对的引用或者指针仍然有效。

C++ STL 标准库为了方便用户更好地管控无序容器底层使用的哈希表存储结构，提供了如下成员方法。

成员方法	功能
bucket_count()	返回当前容器底层存储键值对时，使用桶的数量。
max_bucket_count()	返回当前系统中，unordered_map 容器底层最多可以使用多少个桶。
bucket_size(n)	返回第 n 个桶中存储键值对的数量。
bucket(key)	返回以 key 为键的键值对所在桶的编号。
load_factor()	返回 unordered_map 容器中当前的负载因子。
max_load_factor()	返回或者设置当前 unordered_map 容器的最大负载因子。
rehash(n)	尝试重新调整桶的数量为等于或大于 n 的值。如果 n 大于当前容器使用的桶数，则该方法会是容器重新哈希，该容器新的桶数将等于或大于 n。反之，如果 n 的值小于当前容器使用的桶数，则调用此方法可能没有任何作用。
reserve(n)	将容器使用的桶数（bucket_count() 方法的返回值）设置为最适合存储 n 个元素的桶数。
hash_function()	返回当前容器使用的哈希函数对象。

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    //创建空 umap 容器
    unordered_map<string, string> umap;

    cout << "umap 初始桶数: " << umap.bucket_count() << endl;
    cout << "umap 初始负载因子: " << umap.load_factor() << endl;
    cout << "umap 最大负载因子: " << umap.max_load_factor() << endl;


    //设置 umap 使用最适合存储 9 个键值对的桶数
    umap.reserve(9);
    cout << "**********" << endl;
    cout << "umap 新桶数: " << umap.bucket_count() << endl;
    cout << "umap 新负载因子: " << umap.load_factor() << endl;
    //向 umap 容器添加 3 个键值对
    umap["Python教程"] = "http://c.biancheng.net/python/";
    umap["Java教程"] = "http://c.biancheng.net/java/";
    umap["Linux教程"] = "http://c.biancheng.net/linux/";
    //调用 bucket() 获取指定键值对位于桶的编号
    cout << "以\"Python教程\"为键的键值对，位于桶的编号为:" << umap.bucket("Python教程") << endl;
    //自行计算某键值对位于哪个桶
    auto fn = umap.hash_function();
    cout << "计算以\"Python教程\"为键的键值对，位于桶的编号为：" << fn("Python教程") % (umap.bucket_count()) << endl;
    return 0;
```

```
}
umap 初始桶数: 8
umap 初始负载因子: 0
umap 最大负载因子: 1
**********
umap 新桶数: 16
umap 新负载因子: 0
以"Python教程"为键的键值对，位于桶的编号为:9
计算以"Python教程"为键的键值对，位于桶的编号为：9
```

从输出结果可以看出，对于空的umap容器，初始状态下会分配8个桶，并且默认最大负载因子为1.0，但由于其未存储任何键值对，因此负载因子值为0。与此同时，程序中reserve()成员方法，使得umap容器的桶数改为16，其最适合存储9个键值对。从侧面可以看出，一旦负载因子大于1.0（9/8>1.0），则容器所使用的桶数会翻倍式（8、16、32、...）地增加。

12.4 unordered_multimap容器用法

12.4.1 容器说明

和 unordered_map 容器一样，unordered_multimap 容器也以键值对的形式存储数据，且底层也采用哈希表结构存储各个键值对。两者唯一的不同之处在于，unordered_multimap 容器可以存储多个键相等的键值对，而 unordered_map 容器不行。

无序容器中存储的各个键值对，都会哈希存到各个桶（本质为链表）中。而对于 unordered_multimap 容器来说，其存储的所有键值对中，键相等的键值对会被哈希到同一个桶中存储。

另外值得一提得是，STL 标准库中实现 unordered_multimap 容器的模板类并没有定义在以自己名称命名的头文件中，而是和 unordered_map 容器一样，定义在<unordered_map>头文件，且位于 std 命名空间中。

12.4.2 创建容器的方法

（1）利用 unordered_multimap 容器类模板中的默认构造函数，可以创建空的 unordered_multimap 容器。比如：

```
std::unordered_multimap<std::string, std::string>myummap;
```

由此，就创建好了一个可存储 <string, string> 类型键值对的 unordered_multimap 容器，只不过当前容器是空的，即没有存储任何键值对。

（2）当然，在创建空 unordered_multimap 容器的基础上，可以完成初始化操作。比如：

```
unordered_multimap<string, string>myummap{
    {"Python教程","http://c.biancheng.net/python/"},
    {"Java教程","http://c.biancheng.net/java/"},
    {"Linux教程","http://c.biancheng.net/linux/"} };
```

（3）另外，unordered_multimap 模板中还提供有复制（拷贝）构造函数，可以实现在创建 unordered_multimap 容器的基础上，用另一 unordered_multimap 容器中的键值对为其初始化。

```
unordered_multimap<string, string>myummap2(myummap);
```

除此之外，C++ 11 标准中还向 unordered_multimap 模板类增加了移动构造函数，即以右值引用的方式将临时 unordered_multimap 容器中存储的所有键值对，全部复制给新建容器。例如：

```
//返回临时 unordered_multimap 容器的函数
std::unordered_multimap <std::string, std::string > retUmmap() {
    std::unordered_multimap<std::string, std::string>tempummap{
        {"Python教程","http://c.biancheng.net/python/"},
        {"Java教程","http://c.biancheng.net/java/"},
        {"Linux教程","http://c.biancheng.net/linux/"} };
    return tempummap;
}
//创建并初始化 myummap 容器
std::unordered_multimap<std::string, std::string> myummap(retummap());
```

（4）当然，如果不想全部拷贝，可以使用 unordered_multimap 类模板提供的迭代器，在现有 unordered_multimap 容器中选择部分区域内的键值对，为新建 unordered_multimap 容器初始化。例如：

```
//传入 2 个迭代器，
std::unordered_multimap<std::string, std::string> myummap2(++myummap.begin(), myummap.end());
```

12.4.3 unordered_multimap容器的成员方法

成员方法	功能
begin()	返回指向容器中第一个键值对的正向迭代器。
end()	返回指向容器中最后一个键值对之后位置的正向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。

empty()	若容器为空，则返回 true；否则 false。
size()	返回当前容器中存有键值对的个数。
max_size()	返回容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。
find(key)	查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如果 end() 方法返回的迭代器）。
count(key)	在容器中查找以 key 键的键值对的个数。
equal_range(key)	返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中键为 key 的键值对所在的范围。
emplace()	向容器中添加新键值对，效率比 insert() 方法高。
emplace_hint()	向容器中添加新键值对，效率比 insert() 方法高。
insert()	向容器中添加新键值对。
erase()	删除指定键值对。
clear()	清空容器，即删除容器中存储的所有键值对。
swap()	交换 2 个 unordered_multimap 容器存储的键值对，前提是必须保证这 2 个容器的类型完全相等。
bucket_count()	返回当前容器底层存储键值对时，使用桶（一个线性链表代表一个桶）的数量。
max_bucket_count()	返回当前系统中，unordered_multimap 容器底层最多可以使用多少桶。
bucket_size(n)	返回第 n 个桶中存储键值对的数量。
bucket(key)	返回以 key 为键的键值对所在桶的编号。
load_factor()	返回 unordered_multimap 容器中当前的负载因子。负载因子，指的是的当前容器中存储键值对的数量（size()）和使用桶数（bucket_count()）的比值，即 load_factor() = size() / bucket_count()。
max_load_factor()	返回或者设置当前 unordered_multimap 容器的负载因子。
rehash(n)	将当前容器底层使用桶的数量设置为 n。
reserve()	将存储桶的数量（也就是 bucket_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。
hash_function()	返回当前容器使用的哈希函数对象。

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;


int main()
{
    //创建空容器
    std::unordered_multimap<std::string, std::string> myummap;
    //向空容器中连续添加 5 个键值对
    myummap.emplace("Python教程", "http://c.biancheng.net/python/");
    myummap.emplace("STL教程", "http://c.biancheng.net/stl/");
    myummap.emplace("Java教程", "http://c.biancheng.net/java/");
    myummap.emplace("C教程", "http://c.biancheng.net");
    myummap.emplace("C教程", "http://c.biancheng.net/c/");
    //输出 muummap 容器存储键值对的个数
    cout << "myummmap size = " << myummap.size() << endl;
    //利用迭代器输出容器中存储的所有键值对
    for (auto iter = myummap.begin(); iter != myummap.end(); ++iter) {
        cout << iter->first << " " << iter->second << endl;
    }
    return 0;
}
myummmap size = 5
Python教程 http://c.biancheng.net/python/
C教程 http://c.biancheng.net
```

12.5 unordered_set容器用法

12.5.1 容器说明

unordered_set 容器，可直译为“无序 set 容器”，即 unordered_set 容器和 set 容器很像，唯一的区别就在于 set 容器会自行对存储的数据进行排序，而 unordered_set 容器不会。

总的来说，unordered_set 容器具有以下几个特性：

- 不再以键值对的形式存储数据，而是直接存储数据的值；
- 容器内部存储的各个元素的值都互不相等，且不能被修改。
- 不会对内部存储的数据进行排序

12.5.2 创建容器的方法

（1）通过调用 unordered_set 模板类的默认构造函数，可以创建空的 unordered_set 容器。比如：

```
std::unordered_set<std::string> uset;
```

由此，就创建好了一个可存储 string 类型值的 unordered_set 容器，该容器底层采用默认的哈希函数 hash<Key> 和比较函数 equal_to<Key>。

（2）当然，在创建 unordered_set 容器的同时，可以完成初始化操作。比如：

```
std::unordered_set<std::string> uset{ "http://c.biancheng.net/c/",
                                      "http://c.biancheng.net/java/",
                                      "http://c.biancheng.net/linux/" };
```

（3）还可以调用 unordered_set 模板中提供的复制（拷贝）构造函数，将现有 unordered_set 容器中存储的元素全部用于为新建 unordered_set 容器初始化。

```
std::unordered_set<std::string> uset2(uset);
```

除此之外，C++ 11 标准中还向 unordered_set 模板类增加了移动构造函数，即以右值引用的方式，利用临时 unordered_set 容器中存储的所有元素，给新建容器初始化。例如：

```
//返回临时 unordered_set 容器的函数
std::unordered_set <std::string> retuset() {
    std::unordered_set<std::string> tempuset{ "http://c.biancheng.net/c/",
                                              "http://c.biancheng.net/java/",
                                              "http://c.biancheng.net/linux/" };
    return tempuset;
}
//调用移动构造函数，创建 uset 容器
std::unordered_set<std::string> uset(retuset());
```

（4）当然，如果不想全部拷贝，可以使用 unordered_set 类模板提供的迭代器，在现有 unordered_set 容器中选择部分区域内的元素，为新建 unordered_set 容器初始化。例如：

```
//传入 2 个迭代器，
std::unordered_set<std::string> uset2(++uset.begin(),uset.end());
```

12.5.3 unordered_set容器的成员方法

成员方法	功能
begin()	返回指向容器中第一个键值对的正向迭代器。
end()	返回指向容器中最后一个键值对之后位置的正向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。
empty()	若容器为空，则返回 true；否则 false。
size()	返回当前容器中存有键值对的个数。
max_size()	返回容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。
find(key)	查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如果 end() 方法返回的迭代器）。
count(key)	在容器中查找以 key 键的键值对的个数。
equal_range(key)	返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中键为 key 的键值对所在的范围。

emplace()	向容器中添加新键值对，效率比 insert() 方法高。
emplace_hint()	向容器中添加新键值对，效率比 insert() 方法高。
insert()	向容器中添加新键值对。
erase()	删除指定键值对。
clear()	清空容器，即删除容器中存储的所有键值对。
swap()	交换 2 个 unordered_set 容器存储的键值对，前提是必须保证这 2 个容器的类型完全相等。
bucket_count()	返回当前容器底层存储键值对时，使用桶（一个线性链表代表一个桶）的数量。
max_bucket_count()	返回当前系统中，unordered_set 容器底层最多可以使用多少桶。
bucket_size(n)	返回第 n 个桶中存储键值对的数量。
bucket(key)	返回以 key 为键的键值对所在桶的编号。
load_factor()	返回 unordered_set 容器中当前的负载因子。负载因子，指的是的当前容器中存储键值对的数量（size()）和使用桶数（bucket_count()）的比值，即 load_factor() = size() / bucket_count()。
max_load_factor()	返回或者设置当前 unordered_set 容器的负载因子。
rehash(n)	将当前容器底层使用桶的数量设置为 n。
reserve()	将存储桶的数量（也就是 bucket_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。
hash_function()	返回当前容器使用的哈希函数对象。

注意，此容器模板类中没有重载 [ ] 运算符，也没有提供 at() 成员方法。不仅如此，由于 unordered_set 容器内部存储的元素值不能被修改，因此无论使用哪个迭代器方法获得的迭代器，都不能用于修改容器中元素的值。

另外，对于实现互换 2 个相同类型 unordered_set 容器的所有元素，除了调用表 2 中的 swap() 成员方法外，还可以使用 STL 标准库提供的 swap() 非成员函数，它们具有相同的名称，用法也相同（都只需要传入 2 个参数即可），仅是调用方式上有差别。

```
#include <iostream>
#include <string>
#include <unordered_set>
using namespace std;


int main()
{
    //创建一个空的unordered_set容器
    std::unordered_set<std::string> uset;
    //给 uset 容器添加数据
    uset.emplace("http://c.biancheng.net/java/");
    uset.emplace("http://c.biancheng.net/c/");
    uset.emplace("http://c.biancheng.net/python/");
    //查看当前 uset 容器存储元素的个数
    cout << "uset size = " << uset.size() << endl;
    //遍历输出 uset 容器存储的所有元素
    for (auto iter = uset.begin(); iter != uset.end(); ++iter) {
        cout << *iter << endl;
    }
    return 0;
}
uset size = 3
http://c.biancheng.net/java/
http://c.biancheng.net/c/
http://c.biancheng.net/python/
```

12.6 unordered_multiset容器用法

12.6.1 容器说明

unordered_multiset 容器大部分的特性都和 unordered_set 容器相同，包括：

- unordered_multiset 不以键值对的形式存储数据，而是直接存储数据的值；
- 该类型容器底层采用的也是哈希表存储结构，它不会对内部存储的数据进行排序；
- unordered_multiset 容器内部存储的元素，其值不能被修改。

和 unordered_set 容器不同的是，unordered_multiset 容器可以同时存储多个值相同的元素，且这些元素会存储到哈希表中同一个桶（本质就是链表）上。

另外值得一提的是，实现 unordered_multiset 容器的模板类并没有定义在以该容器名命名的文件中，而是和 unordered_set 容器共用同一个<unordered_set>头文件，并且也位于 std 命名空间。

12.6.2 创建容器的方法

（1）调用 unordered_multiset 模板类的默认构造函数，可以创建空的 unordered_multiset 容器。比如：

```
std::unordered_multiset<std::string> umset;
```

由此，就创建好了一个可存储 string 类型值的 unordered_multiset 容器，该容器底层采用默认的哈希函数 hash<Key> 和比较函数 equal_to<Key>。

（2）当然，在创建 unordered_multiset 容器的同时，可以进行初始化操作。比如：

```
std::unordered_multiset<std::string> umset{ "http://c.biancheng.net/c/",
                                            "http://c.biancheng.net/java/",
                                            "http://c.biancheng.net/linux/" };
```

（3）还可以调用 unordered_multiset 模板中提供的复制（拷贝）构造函数，将现有 unordered_multiset 容器中存储的元素全部用于为新建 unordered_multiset 容器初始化。

```
std::unordered_multiset<std::string> umset2(umset);
```

除此之外，C++ 11 标准中还向 unordered_multiset 模板类增加了移动构造函数，即以右值引用的方式，利用临时 unordered_multiset 容器中存储的所有元素，给新建容器初始化。例如：

```
//返回临时 unordered_multiset 容器的函数
std::unordered_multiset <std::string> retumset() {
    std::unordered_multiset<std::string> tempumset{ "http://c.biancheng.net/c/",
                                                    "http://c.biancheng.net/java/",
                                                    "http://c.biancheng.net/linux/" };
    return tempumset;
}
//调用移动构造函数，创建 umset 容器
std::unordered_multiset<std::string> umset(retumset());
```

（4）当然，如果不想全部拷贝，可以使用 unordered_multiset 类模板提供的迭代器，在现有 unordered_multiset 容器中选择部分区域内的元素，为新建 unordered_multiset 容器初始化。例如：

```
//传入 2 个迭代器，
std::unordered_multiset<std::string> umset2(++umset.begin(), umset.end());
```

12.6.3 unordered_multiset容器的成员方法

成员方法	功能
begin()	返回指向容器中第一个键值对的正向迭代器。
end()	返回指向容器中最后一个键值对之后位置的正向迭代器。
cbegin()	和 begin() 功能相同，只不过在其基础上增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。
empty()	若容器为空，则返回 true；否则 false。
size()	返回当前容器中存有键值对的个数。
max_size()	返回容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。
find(key)	查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如果 end() 方法返回的迭代器）。
count(key)	在容器中查找以 key 键的键值对的个数。
equal_range(key)	返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中键为 key 的键值对所在的范围。
emplace()	向容器中添加新键值对，效率比 insert() 方法高。

emplace_hint()	向容器中添加新键值对，效率比 insert() 方法高。
insert()	向容器中添加新键值对。
erase()	删除指定键值对。
clear()	清空容器，即删除容器中存储的所有键值对。
swap()	交换 2 个 unordered_multiset 容器存储的键值对，前提是必须保证这 2 个容器的类型完全相等。
bucket_count()	返回当前容器底层存储键值对时，使用桶（一个线性链表代表一个桶）的数量。
max_bucket_count()	返回当前系统中，unordered_set 容器底层最多可以使用多少桶。
bucket_size(n)	返回第 n 个桶中存储键值对的数量。
bucket(key)	返回以 key 为键的键值对所在桶的编号。
load_factor()	返回 unordered_multiset 容器中当前的负载因子。负载因子，指的是的当前容器中存储键值对的数量（size()）和使用桶数（bucket_count()）的比值，即 load_factor() = size() / bucket_count()。
max_load_factor()	返回或者设置当前 unordered_multiset 容器的负载因子。
rehash(n)	将当前容器底层使用桶的数量设置为 n。
reserve()	将存储桶的数量（也就是 bucket_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。
hash_function()	返回当前容器使用的哈希函数对象。

注意，和 unordered_set 容器一样，unordered_multiset 模板类也没有重载 [ ] 运算符，没有提供 at() 成员方法。不仅如此，无论是由哪个成员方法返回的迭代器，都不能用于修改容器中元素的值。

另外，对于互换 2 个相同类型 unordered_multiset 容器存储的所有元素，除了调用表 2 中的 swap() 成员方法外，STL 标准库也提供了 swap() 非成员函数。

```
#include <iostream>
#include <string>
#include <unordered_set>
using namespace std;
int main()
{
    //创建一个空的unordered_multiset容器
    std::unordered_multiset<std::string> umset;
    //给 uset 容器添加数据
    umset.emplace("http://c.biancheng.net/java/");
    umset.emplace("http://c.biancheng.net/c/");
    umset.emplace("http://c.biancheng.net/python/");
    umset.emplace("http://c.biancheng.net/c/");
    //查看当前 umset 容器存储元素的个数
    cout << "umset size = " << umset.size() << endl;
    //遍历输出 umset 容器存储的所有元素
    for (auto iter = umset.begin(); iter != umset.end(); ++iter) {
        cout << *iter << endl;
    }
    return 0;
}
umset size = 4
http://c.biancheng.net/java/
http://c.biancheng.net/c/
http://c.biancheng.net/c/
http://c.biancheng.net/python/
```

来自 <https://blog.csdn.net/u014465639/article/details/70241850>