

C++标准库 (Part1)

写在前面：这篇笔记是 侯捷STL源码剖析 教学视频的学习笔记，内容与侯捷自己编撰的《STL源码剖析》一书应该有部分相同，这个教学视频大概是2015年左右的，有部分内容已经过时（比如所有insert函数现在都有了对应的emplace函数），但并不妨碍初学者借此入门STL。

知乎上有关《STL源码剖析》过时的讨论：[《STL源码剖析》和《Effective STL》这两本书的内容是否有些过时？ - 知乎 \(zhihu.com\)](#)

头文件

旧式头文件：`#include <stdio.h>`

新式头文件：`#include <cstdio>` (不带.h, 开头添加c)

新式头文件 (包括 `<vector>` 等)都被封装在 命名空间std 中

STL六大部件

容器(Container) 分配器(Allocator) 迭代器(Iterator)

算法(Algorithm) 适配器(Adapter) 仿函数(Functor)

```
#include <algorithm>
#include <vector>

int ia[6]={1,11,22,13,24,5}; // *ia=1, *(ia+5)=5
vector<int,allocator<int>> vec(ia,ia+6); //传入begin地址和end地址以赋予初值
count_if(vec.begin(),vec.end(),bind2nd(less<int>(),10));

//vector : container
//allocator<int>: allocator, 用于分配内存, 一般可省略
//count_if : algorithm
//vec.begin: iterator
//bind2nd: function adapter
//less<int>(): functor
```

容器结构与分类

- Sequence Container 序列式容器

- 例如: `array` (固定长度序列) `vector` (可扩容序列) `deque` (双向队列) `list`(双向链表) `forward-list` (单向链表)
- `Associative Container` 关联式容器
 - 例如: `set` `map` `multiset` `multimap`
 - `multi` 表示key值 允许重复
 - `set/map` 底层一般是红黑树
- `Unordered Container` 无序式容器
 - 例如: `unordered_set/multiset` `unordered_map/multimap`
 - `unordered`容器通过HashTable实现, 采用分离链接法(链地址法)解决冲突

简单介绍下上面提到的几个容器

- `array`: 固定长度, 提供`data()`函数返回起始地址

```
const int size = 500000;
array<int,size> c;
c.size();
c.front();
c.back();
c.data(); //返回array初始地址
```

- `vector`: 可扩容, 倍增式扩容, `size`一定小于等于`capacity`

```
vector<int> c;
c.push_back(elem);
c.size(); //当前元素数
c.front();
c.back();
c.data(); //返回起始地址
c.capacity(); //空间容量
```

- `list`: 双向链表, 可扩容, 每次扩增一个节点

```
list<string> c;
c.push_back();
c.size(); c.max_size(); c.front(); c.back();
c.sort(); //容器内置sort函数, 一般都比通用的sort函数好, 优先使用容器的sort函数
```

- `forward_list`: 单向链表

```
forward_list<string> c;
c.push_front(); //从前插入
c.max_size(); c.front(); //不提供size()、back()
c.sort(); //也有内置sort
```

- `deque` : 双向队列, 分段连续, `deque`中存储指向多个连续buffer的指针, 每次扩容扩充一个buffer大小

```
push_back(), back(), size(), max_size(), front()
```

- `stack` 和 `queue` 都是通过 `queue` 实现的, 故 `stack` 和 `queue` 属于 `container adapter`

```
//stack
push() size() top() pop()

//queue
push() front() back() size() pop()
```

- `stack` 和 `queue` 不提供 `iterator`操作
- 以下是容器间的复合关系:
 - `set`拥有`rb tree` (红黑树)
 - `heap`拥有`vector`
 - `queue/stack`拥有`deque`
 - `priority_queue`拥有`heap`

关联式容器查找更快、插入也快

- `multiset` : 允许插入重复元素的集合

```
multiset<string> c;
c.insert(elem); //insert函数插入,set/map容器都采用insert函数插入
c.find(target); //提供内置的find函数, 返回iterator
c.size(); c.max_size();
```

- `multimap` : 不可以用 `[]` 做插入

```

multimap<long,string> c;
c.insert(pair<long,string>(key,value));
c.find(target); //提供内置的find函数, 返回iterator
c.size(); c.max_size();

```

- `set` : 同multiset, 但放入重复元素会跳过
- `map` : 同multimap, 但可以用 `[]` 做插入, 例如: `c[key]=value;`
- `unordered_multiset` : 基于hashTable实现的set

```

c.insert(key);
c.size(); c.max_size();
c.bucket_count(); //bucket_count比size大, 因为链地址法, 有些桶是空的
c.load_factor(); c.max_load_factor(); c.max_bucket_count();
c.find(target);
c.bucket_size(i); //可查询某个桶有多少元素

```

- `unordered_multimap` : 基于hashTable实现的map

分配器 (allocator)

默认的容器分配器都是 `std::allocator`

```

#include <memory> //内含 std::allocator
//欲使用 std::allocator 以外的 allocator, 得自行 #include <ext\...>
#ifdef __GNUC__
#include <ext/array_allocator.h>
#include <ext/mt_allocator.h>
#include <ext/debug_allocator.h>
#include <ext/pool_allocator.h> //针对内存分配时cookie带来的额外开销做了优化
#include <ext/bitmap_allocator.h>
#include <ext/malloc_allocator.h>
#include <ext/new_allocator.h>
#endif

list<string, allocator<string>> c1;
list<string, __gnu_cxx::malloc_allocator<string>> c2;
list<string, __gnu_cxx::new_allocator<string>> c3;
list<string, __gnu_cxx::__pool_alloc<string>> c4;
list<string, __gnu_cxx::__mt_alloc<string>> c5;
list<string, __gnu_cxx::bitmap_allocator<string>> c6;

```

直接使用 `allocator` (不推荐)

```
int* p;
allocator<int> alloc1;
p = alloc1.allocate(1); //allocate调用operator new,operator new 调用malloc
alloc1.deallocate(p,1); //deallocate调用operator delete,operator delete调用free

__gnu_cxx::malloc_allocator<int> alloc2;
p = alloc2.allocate(1);
alloc2.deallocate(p,1);
```

泛型编程 Generic Programming

OOP(ObjectOrientedProgramming) : 将 data 和 method 合并

GP(Generic Programming) : 将 data 和 method 分开, 标准库基于GP设计, 好处是便于分开设计 容器 和 算法, 容器 和 算法 通过 iterator 沟通

list 为什么不能用通用的sort()函数? (list内置sort函数)

因为 sort函数 要求具有 随机访问性的迭代器 (random access iterator), 而list容器的iterator只能顺序访问

STL各家 (MSVC、GCC) 实现都不一样, 各个版本可能差距也很大, 下面的探讨旨在说明大致的设计思想, 代码以GCC10.3.0的STL为主

list

- list 的成员应该是一个Node指针, 而 list的迭代器 应该实现相关的运算符重载

```
//对list容器, 其成员包含一个node指针,
template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
class list{
    typedef _List_iterator<_Tp> iterator; //迭代器
    typedef _List_node<_Tp> _Node; //双向链表的节点, 至少包含prev、next、data
    ...
}
```

```
//_List_iterator
template<typename _Tp>
struct _List_iterator
{
```

```

typedef _List_iterator<_Tp>          _Self;
typedef _List_node<_Tp>              _Node;

typedef ptrdiff_t                    difference_type;
typedef std::bidirectional_iterator_tag iterator_category;
typedef _Tp                          value_type;
typedef _Tp*                         pointer;
typedef _Tp&                         reference;

// Must downcast from _List_node_base to _List_node to get to value.
reference
operator*() const // *iterator 等同于 取元素
{ return *static_cast<_Node*>(_M_node)->_M_valptr(); }
// return (*node).data; 上面一行代码等同于这个

pointer
operator->() const
{ return static_cast<_Node*>(_M_node)->_M_valptr(); }
// return &(operator*()); 上面一行代码等同于这个
// iterator->method() = (*iterator).method() = &(*iterator)->method()

_Self&
operator++() // 前++无参数
{
    _M_node = _M_node->_M_next;
    return *this;
}

_Self
operator++(int) // 后++有一个参数
{
    _Self __tmp = *this;
    _M_node = _M_node->_M_next;
    return __tmp;
}

...
}

```

```

// List_node, 包含prev、next、data
struct _List_node_base
{
    _List_node_base* _M_next;
    _List_node_base* _M_prev;
    ...
};

template<typename _Tp>
struct _List_node : public _List_node_base
{
    #if __cplusplus >= 201103L

```

```

__gnu_cxx::__aligned_membuf<_Tp> _M_storage;
_Tp* _M_valptr() { return _M_storage._M_ptr(); }
_Tp const* _M_valptr() const { return _M_storage._M_ptr(); }
#else
_Tp _M_data;
_Tp* _M_valptr() { return std::__addressof(_M_data); }
_Tp const* _M_valptr() const { return std::__addressof(_M_data); }
#endif
};

```

- 为了符合 前闭后开 的设计原则，所以 最后一个节点 指向的下一个节点是 空节点， 第一个节点 指向的前一个节点也是这个 空节点

iterator traits 萃取机

- iterator 是沟通算法和容器的桥梁，算法需要通过iterator知道容器的一些属性，包括 5种
 - iterator_category：迭代器的类型
 - pointer：迭代器中元素的指针
 - reference：迭代器中元素的引用
 - value_type：迭代器中元素的类型
 - difference_type：两个迭代器之间的距离

```

//_List_iterator
template<typename _Tp>
struct _List_iterator
{
    typedef ptrdiff_t difference_type;
    typedef std::bidirectional_iterator_tag iterator_category;
    typedef _Tp value_type;
    typedef _Tp* pointer;
    typedef _Tp& reference;
    ...
}

```

- iterator_traits 作为 中间层，防止 iterator 不是 class 而是 native pointer 的情况下，算法中直接调用 iterator::iterator_category 获取属性出错

```

template<class I>
struct iterator_traits{ //如果iterator是class 进入此处
    typedef typename I::value_type value_type;
}

template<class T>
struct iterator_traits<T*>{ //偏特化，如果iterator是pointer to T 进入此处

```

```

    typedef T value_type;
}

template<class T>
struct iterator_traits<const T*>{ //偏特化, 如果iterator是pointer to const T 进入此处
    typedef T value_type;
}

```

```

//使用iterator_traits
template<typename I,...>
void algorithm(){
    typename iterator_traits<I>::value_type v1;
}

```

- 除了iterator traits外, 还有各种各样的traits : type/char/pointer/allocator/array traits

vector

三个iterator: start、finish、end_of_storage

每次扩容都是 双倍扩容, 需要拷贝原vector到新的vector

array

array 是 定长数组, 最小长度为1, 用 指针 当迭代器

```

template<typename _Tp, std::size_t _Nm>
struct array
{
    typedef _Tp value_type;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef value_type* iterator;
    ...
}

```

deque

- deque的实现原理: 用连续的存储空间存储指针 (map), 每个指针指向一段具有连续空间的buffer
 - deque包含的成员: map(指向buffer的指针), map_size, start iterator, finish iterator


```

* Here's how a deque<Tp> manages memory. Each deque has 4 members:
* - Tp**      _M_map
* - size_t    _M_map_size
* - iterator  _M_start, _M_finish

typedef _Deque_iterator<Tp, _Tp&, _Ptr>      iterator;
typedef typename iterator::_Map_pointer _Map_pointer; //见下面deque_iterator中定义

_Map_pointer _M_map;
size_t _M_map_size;
iterator _M_start;
iterator _M_finish;

```

- deque的iterator: 包含cur/first/last/node
 - cur指向当前buffer中当前位置
 - first指向buffer的起始点
 - last指向一个buffer的终点
 - node指向当前的buffer, 也就是map中的一个节点

```

typedef _Tp*      _Elt_pointer;
typedef _Tp**     _Map_pointer;

_Elt_pointer _M_cur;
_Elt_pointer _M_first;
_Elt_pointer _M_last;
_Map_pointer _M_node;

```

- deque的 iterator 需要实现各种 iterator运算操作, 保证在各个不连续buffer中移动, 目的是 模拟连续空间

```

_Self&
operator++() _GLIBCXX_NOEXCEPT //前++
{
    ++_M_cur;
    if (_M_cur == _M_last) //iterator到了buffer的末尾, 前闭后开, 所以跳转到下一个buffer
    {
        _M_set_node(_M_node + 1); //跳到下一个buffer
        _M_cur = _M_first;
    }
    return *this;
}

void
_M_set_node(_Map_pointer __new_node) _GLIBCXX_NOEXCEPT
{
    _M_node = __new_node;
    _M_first = *__new_node;
}

```

```

        _M_last = _M_first + difference_type(_S_buffer_size());
    }

    _Self&
    operator++(int) _GLIBCXX_NOEXCEPT //后++
    {
        _Self __tmp = *this;
        ++*this;
        return __tmp;
    }
    ...

```

```

    _Self&
    operator--() _GLIBCXX_NOEXCEPT //--
    {
        if (_M_cur == _M_first)
        {
            _M_set_node(_M_node - 1);
            _M_cur = _M_last;
        }
        --_M_cur;
        return *this;
    }

    _Self&
    operator+=(difference_type __n) _GLIBCXX_NOEXCEPT //+=
    {
        const difference_type __offset = __n + (_M_cur - _M_first);
        if (__offset >= 0 && __offset < difference_type(_S_buffer_size()))
            _M_cur += __n;
        else
        {
            const difference_type __node_offset =
                __offset > 0 ? __offset / difference_type(_S_buffer_size())
                : -difference_type((-__offset - 1)
                    / _S_buffer_size()) - 1;
            _M_set_node(_M_node + __node_offset);
            _M_cur = _M_first + (__offset - __node_offset
                * difference_type(_S_buffer_size()));
        }
        return *this;
    }
}

```

deque的insert操作:

1. 先是判断是否是头插或者尾插。是的话直接头尾插入元素即可。2. 如果不是头插或者尾插，那么计算这个节点到头结点和尾节点之间的距离。假如说离头部节点近，那么就头节点到插入位置之间的节点全部向前挪动，然后插入节点；反之亦然。

queue和stack

queue和stack都是默认用 deque 作为 底层容器，但也可以用其他的容器作为底层容器

```
stack<int, list<int>> st; //以list为底层容器的写法
```

1、stack可以用 deque、list、vector 作为底层容器

2、queue可以用 deque、list、作为底层容器

3、stack/queue都 不 可以用map/set作为底层容器

stack/queue不允许遍历，不提供iterator

rb_tree

map/set 包含（复合）了 rb_tree，rb_tree包含 rb_tree_impl，rb_tree_impl包含 _Rb_tree_node_base

rb_tree作为容器，提供 iterator

```
template<typename _Tp>
struct _Rb_tree_iterator
{
    typedef _Tp value_type;
    typedef _Tp& reference;
    typedef _Tp* pointer;
    typedef bidirectional_iterator_tag iterator_category;
    typedef ptrdiff_t difference_type;
    typedef _Rb_tree_iterator<_Tp> _Self;
    typedef _Rb_tree_node_base::_Base_ptr _Base_ptr;
    typedef _Rb_tree_node<_Tp>* _Link_type;

    _Base_ptr _M_node;
};
```

```

enum _Rb_tree_color { _S_red = false, _S_black = true };
struct _Rb_tree_node_base
{
    typedef _Rb_tree_node_base* _Base_ptr;

    _Rb_tree_color    _M_color;
    _Base_ptr         _M_parent;
    _Base_ptr         _M_left;
    _Base_ptr         _M_right;
    ...
}

```

```

template<typename _Key, typename _Val, typename _KeyOfValue,
typename _Compare, typename _Alloc = allocator<_Val> > //5个模板参数
class _Rb_tree
{
protected:
    typedef _Rb_tree_node_base*      _Base_ptr;
    typedef _Rb_tree_node<_Val>*    _Link_type;

private:
    _Base_ptr _M_root;
    _Base_ptr _M_nodes;
    _Rb_tree& _M_t;

    _Rb_tree_impl<_Compare> _M_impl; //继承_Rb_tree_header、_Rb_tree_key_compare等
public:
    typedef _Rb_tree_iterator<value_type>    iterator;
};

```

```

struct _Rb_tree_impl
: public _Node_allocator
    , public _Rb_tree_key_compare<_Key_compare>
    , public _Rb_tree_header
{
    ...
}

// Helper type to manage default initialization of node count and header.
struct _Rb_tree_header
{
    _Rb_tree_node_base _M_header;
    size_t             _M_node_count; // Keeps track of size of tree.
    ...
}

```

```
// Helper type offering value initialization guarantee on the compare functor.
template<typename _Key_compare>
struct _Rb_tree_key_compare
{
    _Key_compare      _M_key_compare;
    ...
}
```

set/multiset

set/multiset都是用rb_tree作为底层容器，它们的 key就是value，不能用迭代器改变set/multiset的key

set的插入调用rb_tree的insert_unique (C++11后insert也可用emplace替代)

multiset的插入调用rb_tree的insert_equal (C++11后insert也可用emplace替代)

```
template<typename _Key, typename _Compare = std::less<_Key>,
typename _Alloc = std::allocator<_Key> >
class set
{
    typedef _Key      key_type;
    typedef _Key      value_type;
    typedef _Compare  key_compare;
    typedef _Compare  value_compare;

    typedef _Rb_tree<key_type, value_type, _Identity<value_type>,
key_compare, _Key_alloc_type> _Rep_type;
    typedef typename _Rep_type::const_iterator  iterator; //const iterator

    _Rep_type _M_t; // Red-black tree representing set

    std::pair<iterator, bool> insert(const value_type& __x)
    {
        std::pair<typename _Rep_type::iterator, bool> __p =
            _M_t._M_insert_unique(__x); //set的insert调用rb_tree的insert_unique
        return std::pair<iterator, bool>(__p.first, __p.second);
    }
    ...
}
```

```
template <typename _Key, typename _Compare = std::less<_Key>,
typename _Alloc = std::allocator<_Key> >
class multiset
```

```

{
    typedef _Key      key_type;
    typedef _Key      value_type;
    typedef _Compare  key_compare;
    typedef _Compare  value_compare;
    typedef _Alloc    allocator_type;

    typedef _Rb_tree<key_type, value_type, _Identity<value_type>,
key_compare, _Key_alloc_type> _Rep_type;
    /// The actual tree structure.
    _Rep_type _M_t;

    //const iterator
    typedef typename _Rep_type::const_iterator    iterator;

    iterator insert(const value_type& __x) //multiset的插入
    { return _M_t._M_insert_equal(__x); } //调用rb_tree的insert_equal
}

```

map/multimap

- map/multimap也是用rb_tree作为底层容器
- 可以用iterator 改变data, 无法改变key
- map/multimap底层的rb_tree的 key和value不同, value 是由 key和data组成的pair
- map 可以通过 operator[] 插入, multimap不行
- 同set和multiset, map和multimap的插入也是分别通过rb_tree的 unique/equal插入函数 实现

```

template <typename _Key, typename _Tp, typename _Compare = std::less<_Key>,
          typename _Alloc = std::allocator<std::pair<const _Key, _Tp> > >
class map
{
public:
    typedef _Key      key_type;
    typedef _Tp      mapped_type;
    typedef std::pair<const _Key, _Tp>      value_type; //value为
(key_type,mapped_type)
    typedef _Compare      key_compare;
private:
    typedef _Rb_tree<key_type, value_type, _Select1st<value_type>,
key_compare, _Pair_alloc_type> _Rep_type;

    /// The actual tree structure.
    _Rep_type _M_t;
}

```

```

mapped_type&
operator[](const key_type& __k) //map独有的operator[], 比直接插入慢, 因为要多调用
lower_bound函数
{
    // concept requirements
    __glibcxx_function_requires(_DefaultConstructibleConcept<mapped_type>)

    iterator __i = lower_bound(__k);
    // __i->first is greater than or equivalent to __k.
    if (__i == end() || key_comp()(__k, (*__i).first))
#if __cplusplus >= 201103L
        __i = _M_t._M_emplace_hint_unique(__i, std::piecewise_construct,
            std::tuple<const key_type&>(__k),
            std::tuple<>());
#else
        __i = insert(__i, value_type(__k, mapped_type()));
#endif
    return (*__i).second;
}

```

HashTable

- SeparateChaining: 分离链接法（链地址法），解决冲突的方法
- bucket: 链地址法中桶的个数，可以选取为一个质数
- rehash: 当单个桶中元素数过多（比如大于桶数）时，查找效率降低，需要重新散列，方式是增加 bucket 数（比如变为原先质数的两倍后附近的质数（新版的STL可能不一样）

```

template <typename _Key, typename _Value, typename _Alloc, typename _ExtractKey,
          typename _Equal, typename _H1, typename _H2, typename _Hash,
          typename _RehashPolicy, typename _Traits>
class _Hashtable{
    * Each _Hashtable data structure has:
    *
    * - _Bucket[]          _M_buckets
    * - _Hash_node_base    _M_before_begin
    * - size_type           _M_bucket_count
    * - size_type           _M_element_count

    * with _Bucket being _Hash_node* and _Hash_node containing:
    *
    * - _Hash_node*        _M_next
    * - Tp                  _M_value
    * - size_t              _M_hash_code if cache_hash_code is true
    *
}

```

```

* @tparam _ExtractKey  Function object that takes an object of type
*   _Value and returns a value of type _Key.
*
* @tparam _Equal  Function object that takes two objects of type k
* and returns a bool-like value that is true if the two objects
* are considered equal.
*
* @tparam _H1  The hash function. A unary function object with
* argument type _Key and result type size_t. Return values should
* be distributed over the entire range [0, numeric_limits<size_t>::max()].
*
* @tparam _H2  The range-hashing function (in the terminology of
* Tavori and Dreizin). A binary function object whose argument
* types and result type are all size_t. Given arguments r and N,
* the return value is in the range [0, N).
*
* @tparam _Hash  The ranged hash function (Tavori and Dreizin). A
* binary function whose argument types are _Key and size_t and
* whose result type is size_t. Given arguments k and N, the
* return value is in the range [0, N). Default: hash(k, N) =
* h2(h1(k), N). If _Hash is anything other than the default, _H1
* and _H2 are ignored.
*
* @tparam _RehashPolicy  Policy class with three members, all of
* which govern the bucket count. _M_next_bkt(n) returns a bucket
* count no smaller than n. _M_bkt_for_elements(n) returns a
* bucket count appropriate for an element count of n.
* _M_need_rehash(n_bkt, n_elt, n_ins) determines whether, if the
* current bucket count is n_bkt and the current element count is
* n_elt, we need to increase the bucket count. If so, returns
* make_pair(true, n), where n is the new bucket count. If not,
* returns make_pair(false, <anything>)
*
* @tparam _Traits  Compile-time class with three boolean
* std::integral_constant members: __cache_hash_code, __constant_iterators,
* __unique_keys.

```

unordered容器

- 底层为hash table

```

/// Base types for unordered_set.
template<bool _Cache>
using __uset_traits = __detail::Hashtable_traits<_Cache, true, true>; //最后参数
为true

template<typename _Value,

```



```

        typename _Hash = hash<_Value>,
        typename _Pred = std::equal_to<_Value>,
        typename _Alloc = std::allocator<_Value>,
        typename _Tr = __uset_traits<__cache_default<_Value, _Hash>::value>>
using __uset_hashtable = _Hashtable<_Value, _Value, _Alloc,
    __detail::_Identity, _Pred, _Hash,
    __detail::_Mod_range_hashing,
    __detail::_Default_ranged_hash,
    __detail::_Prime_rehash_policy, _Tr>;

/// Base types for unordered_multiset.
template<bool _Cache>
using __umset_traits = __detail::_Hashtable_traits<_Cache, true, false>; //最后参
数为false

template<typename _Value,
        typename _Hash = hash<_Value>,
        typename _Pred = std::equal_to<_Value>,
        typename _Alloc = std::allocator<_Value>,
        typename _Tr = __umset_traits<__cache_default<_Value, _Hash>::value>>
using __umset_hashtable = _Hashtable<_Value, _Value, _Alloc,
    __detail::_Identity, _Pred, _Hash,
    __detail::_Mod_range_hashing,
    __detail::_Default_ranged_hash,
    __detail::_Prime_rehash_policy, _Tr>;

```

* @tparam _Value Type of key objects.
 * @tparam _Hash Hashing function object type, defaults to hash<_Value>.
 * @tparam _Pred Predicate function object type, defaults
 * to equal_to<_Value>.
 * @tparam _Alloc Allocator type, defaults to allocator<_Key>.

```

template<typename _Value,
        typename _Hash = hash<_Value>,
        typename _Pred = equal_to<_Value>,
        typename _Alloc = allocator<_Value>>
class unordered_set
{
    // __uset_hashtable
    typedef __uset_hashtable<_Value, _Hash, _Pred, _Alloc> _Hashtable;
    _Hashtable _M_h;

    /**
     * @brief Default constructor creates no elements.
     * @param __n Minimal initial number of buckets.
     * @param __hf A hash functor.
     * @param __eqf A key equality functor.
     * @param __a An allocator object.
     */
    explicit

```

```

unordered_set(size_type __n,
              const hasher& __hf = hasher(),
              const key_equal& __eq = key_equal(),
              const allocator_type& __a = allocator_type())
: _M_h(__n, __hf, __eq, __a)
{ }
}

```

```

template<typename _Value,
         typename _Hash = hash<_Value>,
         typename _Pred = equal_to<_Value>,
         typename _Alloc = allocator<_Value>>
class unordered_multiset
{
    // __umset_hashtable
    typedef __umset_hashtable<_Value, _Hash, _Pred, _Alloc> _Hashtable;
    _Hashtable _M_h;

    /**
     * @brief Default constructor creates no elements.
     * @param __n Minimal initial number of buckets.
     * @param __hf A hash functor.
     * @param __eq A key equality functor.
     * @param __a An allocator object.
     */
    explicit
    unordered_multiset(size_type __n,
                      const hasher& __hf = hasher(),
                      const key_equal& __eq = key_equal(),
                      const allocator_type& __a = allocator_type())
: _M_h(__n, __hf, __eq, __a) { }
}

```

- hash为unordered关联式容器的特化

```

/**
 * struct _Hashtable_traits
 *
 * Important traits for hash tables.
 *
 * @tparam _Cache_hash_code Boolean value. True if the value of
 * the hash function is stored along with the value. This is a
 * time-space tradeoff. Storing it may improve lookup speed by
 * reducing the number of times we need to call the _Hash or _Equal
 * functors.
 *
 * @tparam _Constant_iterators Boolean value. True if iterator and

```

```

* const_iterator are both constant iterator types. This is true
* for unordered_set and unordered_multiset, false for
* unordered_map and unordered_multimap.
*
* @tparam _Unique_keys Boolean value. True if the return value
* of _Hashtable::count(k) is always at most one, false if it may
* be an arbitrary number. This is true for unordered_set and
* unordered_map, false for unordered_multiset and
* unordered_multimap.
*/
//_Unique_keys用于区分是否为可重复key的容器
template<bool _Cache_hash_code, bool _Constant_iterators, bool _Unique_keys>
struct _Hashtable_traits
{
    using __hash_cached = __bool_constant<_Cache_hash_code>;
    using __constant_iterators = __bool_constant<_Constant_iterators>;
    using __unique_keys = __bool_constant<_Unique_keys>;
};

/// unordered_map and unordered_set specializations.
template<typename _Key, typename _Value, typename _Alloc,
        typename _ExtractKey, typename _Equal,
        typename _H1, typename _H2, typename _Hash,
        typename _RehashPolicy, typename _Traits>
struct _Equality<_Key, _Value, _Alloc, _ExtractKey, _Equal,
                _H1, _H2, _Hash, _RehashPolicy, _Traits, true> //多了个true
{
    using __hashtable = _Hashtable<_Key, _Value, _Alloc, _ExtractKey, _Equal,
                                    _H1, _H2, _Hash, _RehashPolicy, _Traits>;

    bool
    _M_equal(const __hashtable&) const;
};

template<typename _Key, typename _Value, typename _Alloc,
        typename _ExtractKey, typename _Equal,
        typename _H1, typename _H2, typename _Hash,
        typename _RehashPolicy, typename _Traits>
bool
_Equality<_Key, _Value, _Alloc, _ExtractKey, _Equal,
        _H1, _H2, _Hash, _RehashPolicy, _Traits, true>::_
_M_equal(const __hashtable& __other) const
{
    using __node_base = typename __hashtable::__node_base;
    using __node_type = typename __hashtable::__node_type;
    const __hashtable* __this = static_cast<const __hashtable*>(this);
    if (__this->size() != __other.size())
        return false;

    for (auto __itx = __this->begin(); __itx != __this->end(); ++__itx)
    {
        std::size_t __ybkt = __other._M_bucket_index(__itx._M_cur);

```

```

__node_base* __prev_n = __other._M_buckets[__ybkt];
if (!__prev_n)
    return false;

for (__node_type* __n = static_cast<__node_type*>(__prev_n->_M_nxt);;
    __n = __n->_M_next())
{
    if (__n->_M_v() == *__itx)
        break;

    if (!__n->_M_nxt
        || __other._M_bucket_index(__n->_M_next()) != __ybkt)
        return false;
}

return true;
}

/// unordered_multiset and unordered_multimap specializations.
template<typename _Key, typename _Value, typename _Alloc,
        typename _ExtractKey, typename _Equal,
        typename _H1, typename _H2, typename _Hash,
        typename _RehashPolicy, typename _Traits>
struct _Equality<_Key, _Value, _Alloc, _ExtractKey, _Equal,
                _H1, _H2, _Hash, _RehashPolicy, _Traits, false> //多了个false
{
    using __hashtable = _Hashtable<_Key, _Value, _Alloc, _ExtractKey, _Equal,
                                    _H1, _H2, _Hash, _RehashPolicy, _Traits>;

    bool
    _M_equal(const __hashtable&) const;
};

template<typename _Key, typename _Value, typename _Alloc,
        typename _ExtractKey, typename _Equal,
        typename _H1, typename _H2, typename _Hash,
        typename _RehashPolicy, typename _Traits>
bool
_Equality<_Key, _Value, _Alloc, _ExtractKey, _Equal,
          _H1, _H2, _Hash, _RehashPolicy, _Traits, false>::
_M_equal(const __hashtable& __other) const
{
    using __node_base = typename __hashtable::__node_base;
    using __node_type = typename __hashtable::__node_type;
    const __hashtable* __this = static_cast<const __hashtable*>(this);
    if (__this->size() != __other.size())
        return false;

    for (auto __itx = __this->begin(); __itx != __this->end(); )
    {
        std::size_t __x_count = 1;

```

```

    auto __itx_end = __itx;
    for (++__itx_end; __itx_end != __this->end()
        && __this->key_eq()(_ExtractKey)(*__itx),
            _ExtractKey)(*__itx_end));
        ++__itx_end)
        ++__x_count;

    std::size_t __ybkt = __other._M_bucket_index(__itx._M_cur);
    __node_base* __y_prev_n = __other._M_buckets[__ybkt];
    if (!__y_prev_n)
        return false;

    __node_type* __y_n = static_cast<__node_type*>(__y_prev_n->_M_nxt);
    for (;;) __y_n = __y_n->_M_next()
    {
        if (__this->key_eq()(_ExtractKey)(__y_n->_M_v()),
            _ExtractKey)(*__itx))
            break;

        if (!__y_n->_M_nxt
            || __other._M_bucket_index(__y_n->_M_next()) != __ybkt)
            return false;
    }

    typename __hashtable::const_iterator __ity(__y_n);
    for (auto __ity_end = __ity; __ity_end != __other.end(); ++__ity_end)
        if (--__x_count == 0)
            break;

    if (__x_count != 0)
        return false;

    if (!std::is_permutation(__itx, __itx_end, __ity))
        return false;

    __itx = __itx_end;
}

return true;
}

```