

STL 六大部件

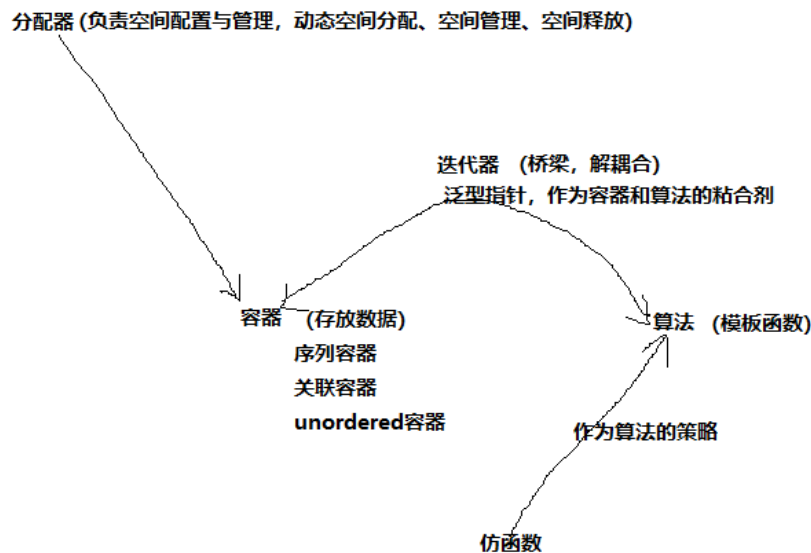
STL 简介

泛型编程 generic programming

将算法和数据结构分别实现：大部分算法被抽象，被泛化，独立于数据结构

模板和偏特化是实现 STL 的核心

程序尽可能通用



适配器：修饰容器 (Containers) 或仿函数 (Functors) 或迭代器 (Iterators) 接口的东西

容器 通过 分配器 取得数据储存空间

算法 通过 迭代器存取 容器 内容

仿函数 可以协助 算法 完成不同的策略变化

适配器 可以修饰或套接 容器, 迭代器, 仿函数

算法看不见容器，它所需要的一切信息都要从迭代器中取得，而迭代器必须能够回答算法的所有提问，才能搭配该算法的所有操作。

STL 预备知识

区间：前闭后开 [)

c.begin() c.end(): 执行容器最后一个元素的下一个元素

遍历容器中的每个元素

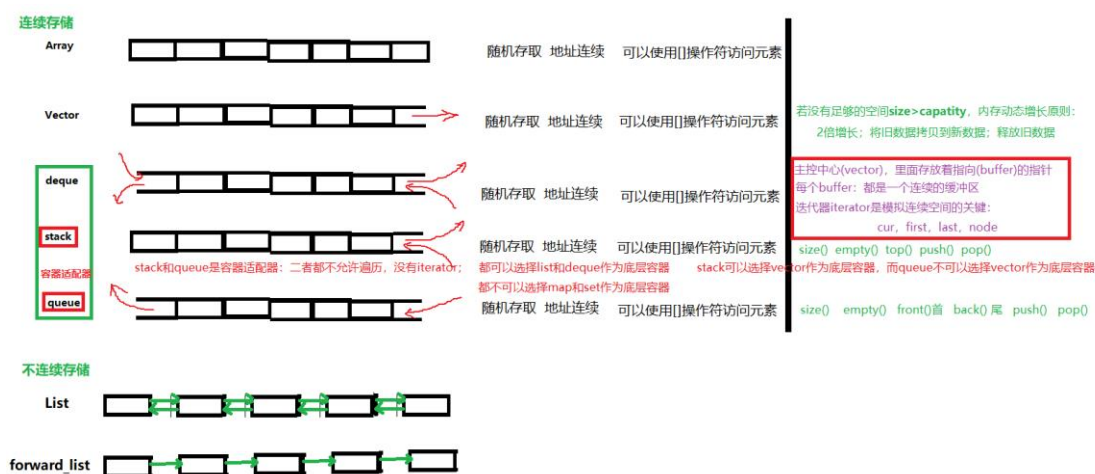
方式一：for(Container<type>::iterator iter = C.begin(); iter!=C.end();iter++) → *iter

方式二：auto 关键字——类型推导

For(auto elem : Containers) → elem

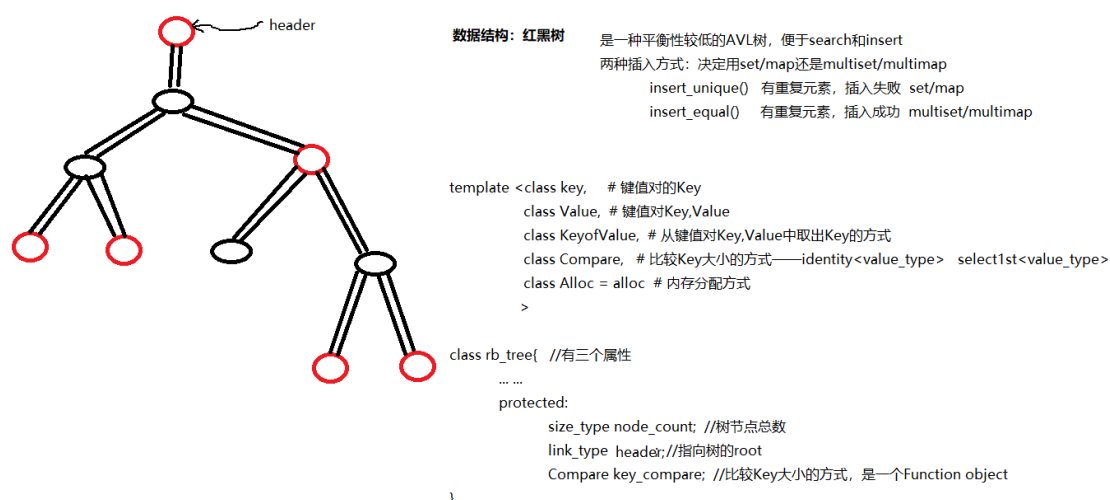
容器 Containers

(1) 序列容器



(2) 关联容器 (rb_tree)

红黑树的特性：按照 iterator 方式遍历，是排好序的 → “元素自动排序”



1-红黑树的性能分析： 查找/插入/删除：log(N)

set/multiset 不能使用迭代器更改 key 的值

map/multimap 不能使用迭代器更改 key 的值，但是可以更改 value 的值

2-介绍 map 的[]特殊的操作符：map[key] = value

如果 key 存在，则返回 key 对应的 value

如果 key 不存在，则将(key,value)键值对安插到 map 中

3-Map 的 insert

```
mapStudent.insert(pair<int, string>(1, "student_one"));
mapStudent[1] = "student_one";
```

4-Map 的反向迭代器: reserver_iterator rbegin() rend() ->first ->second

```
1. map<int, string>::reverse_iterator iter;
2.   for(iter = mapStudent.rbegin(); iter != mapStudent.rend(); iter++)
3.       cout<<iter->first<<" + "<<iter->second<<endl;
```

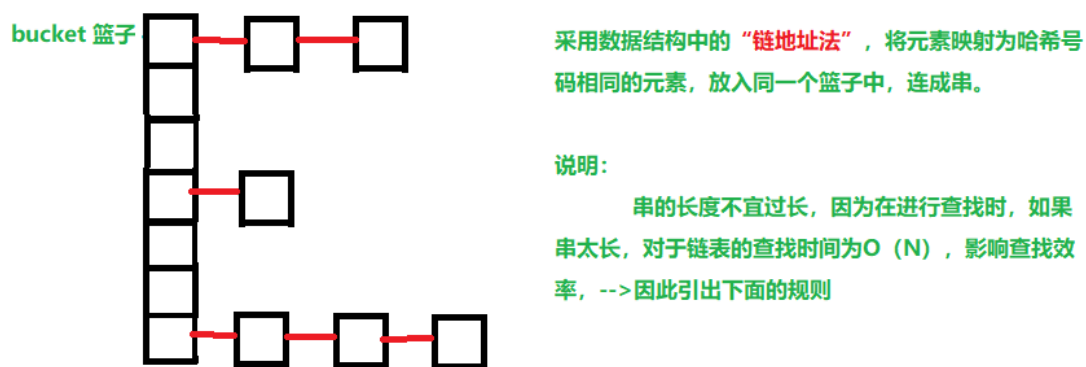
5-Map 的三种数据查找元素，并返回迭代器位置： 迭代器=find(key)

```
1. map<int, string>::iterator iter = mapStudent.find(1);
```

6-Map 的 erase 方法

Iterator erase(iterator iter) //删除 iter 指向的元素，返回值 Iterator 指向 iter 的后继
Iterator erase(iterator first, iterator last) //删除一个范围：前开后闭[first,last)
size_type erase(const Key& key) //通过关键字删除，返回值是 0 或 1

(3) Unordered 容器 (HashTable) 无序



规则：元素个数 > bucket_count(), 就要将 bucket 个数进行扩充，方式见下：

(1) bucket 个数扩充为原来两倍 (2) 重新将元素放入新的 bucket (3) 释放旧篮子空间

面试题型 1：简述 STL 概念（非常重要）

答：

(1)

泛型编程；
程序通用性；

(2)

六大部件的简介和交互（特别是迭代器将容器[数据]和算法[操作]解耦）

面试题型 2：红黑树 哈希表

1. 红黑树的特性与其在 C++ STL 中的应用

关联容器：set, multiset map, multimap

2. map 和 set 的 3 个问题。

（1）为何 map 和 set 的插入删除效率比用其他序列容器 vector，deque 高？

因为对于**关联容器**来说，底层实现是红黑树，插入删除时，**不需要做内存拷贝和内存移动**。

（2）为何 map 和 set 每次 Insert 之后，以前保存的 iterator 不会失效？

因为插入操作只是结点指针换来换去，结点内存没有改变。而 iterator 就像指向结点的指针，内存没变，指向内存的指针也不会变。

（3）当数据元素增多时（从 10000 到 20000），map 的 set 的查找速度会怎样变化？

RB-TREE 用二分查找法，时间复杂度为 $\log n$ ，所以从 10000 增到 20000 时，查找次数从 $\log 10000=14$ 次到 $\log 20000=15$ 次，多了 1 次而已。

2. STL 源码中的 hash table 的实现

3. STL 的 map 和 unordered_map 的区别

（1）map

内部实现了一个**红黑树**，

元素是**自动排序**

查找，插入，删除时间复杂度 $O(\log N)$

（2）unordered_map

内部实现了一个**哈希表**

元素是**无序的**

查找时间复杂度 $O(1)$

面试题型 3：Vector 考点

1. Vector 内存分配与扩容（vector 的扩容机制，问到了源码层次）

先申请一定的大小的数组，当数组填满之后，另外申请一块原数组两倍大的新数组，然后把原数组的数据拷贝到新数组，最后释放原数组的大小。

```

/*
else //如果没有足够的空间--->需要扩容
{
    const size_type old_size = size(); //查看当前vector空间大小old_size
    const size_type len = old_size != 0 ? 2 * old_size : 1; //扩容原则:
    ... //old_size == 0, 则分配1
    //old_size != 0, 则分配为原来的2倍
}
*/

```

2. Vecotr 和 List 的区别和使用场景

区别

动态数组；双向链表

存取方式：随机存取，通过索引查找

连续存储：

vector 在进行插入和删除时，有元素的移动，时间复杂度为 $O(N)$

list 可以进行高效的插入和删除

当 **vector** 空间不够时，需要重新申请一块内存空间

插入和删除数据：

vector 需要对现有数据进行复制移动，如果 **vector** 存储的对象很大或者构造函数很复杂，则开销较大，如果是简单的小数据，效率优于 **list**

list 需要对现有数据进行遍历，但在首部和尾部插入和删除数据，效率很高。

使用场景

高效的随机存取，而不在于插入和删除的效率，使用 **vector**；

大量的插入和删除，而不关心随机存取，则应使用 **list**。

3. Vector 存入大量的数据，两种方式性能对比

(1)不断的 **push_back()**:不断的扩容，需要内存申请，拷贝，释放，消耗 CPU

(2)先申请预备空间 **V.reserve(capacity)**在进行 **push_back**

知识补充：**vector** 的函数 **resize** 和 **reserve** 二者的区别

```

int main()
{
    vector<int> V; //size=capacity=0
    V.resize(10); //size=capacity=10 ,并且V[0]~V[9]已经有值，默认为0 (resize不但进行扩容capacity, 还赋初始值)
    V.reserve(12); //size=10, capacity=12,V[10]和V[11]没有值 (reserve只负责扩容capacity)
}

```

4. 强制释放 vector 的缓冲区——swap 技法 (effective STL)

由于 **vector** 的内存占用空间 **capacity** 只增不减,比如你首先分配了 10,000 个字节,然后 **erase** 掉后面 9,999 个,留下一个有效元素,但是内存占用 **capacity** 仍为 10,000 个。

所有内存空间是在 **vector** 析构时候才能被系统回收。**empty()**用来检测容器是否存在元素；**clear()**可以清空所有元素,但是即使 **clear()**, **vector** 所占用的内存空间 **capacity** 依然不变,无法保证内存的回收。

```

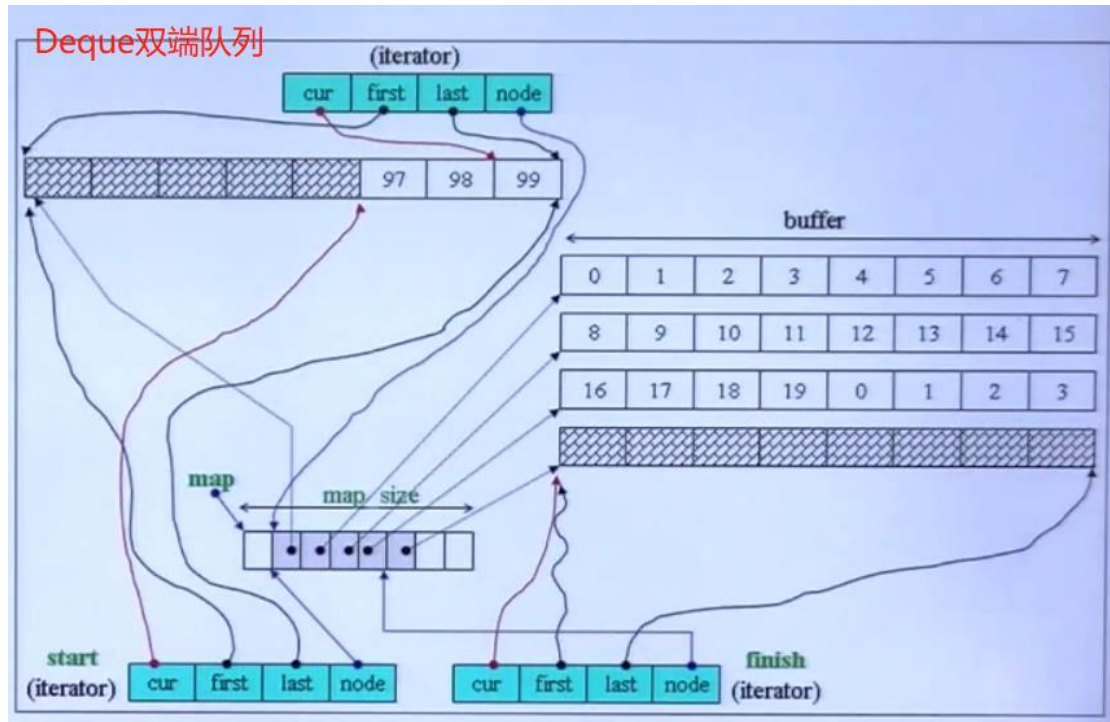
vector<int> V;
vector<int>().swap(V); //或者 V.swap(vector<int> ())
经过 swap()后, size=capacity=0

```

解释：
swap 技法就是通过交换函数 swap()，使 vector 离开自身的作用域，从而强制释放 vector 的内存空间。

面试题 4：双端队列

双端队列的数据的存储形式：主控中心，缓冲区 buffer



Question: 令人疑惑的是——为什么上图中 3 个迭代器类型的指针？

Answer:

每个容器都有 **begin()**和 **end()**函数，指向首部和尾部；因此 **begin()**调用 **start.cur** 指针，**end()**调用 **finish.cur** 指针；

另外一个迭代器（是用于遍历 `deque` 的迭代器）：`iterator` 有四个成员

```
T* cur;           // 指向当前缓冲区中的元素
T* first;         // 当前缓冲区的起点
T* last;          // 当前缓冲区的终点
map_pointer node; // 指向管控中心，用于越界时，进行跳转
```

Deque

- (1) **主控中心**: 中的元素是指针, 指向每一个 **缓冲区 buffer**
- (2) 维护着 **start** 和 **finish** 两个迭代器, 分别指向 **deque** 的首尾
- (3) **内存分配**: 当内存不够时, 主控中心向左或向右增加一个元素 (该元素是指针), 指针再指向一个新开辟一个 **buffer**。

面试题 5: C++ STL 迭代器失效问题

1. 对于 vector 和 string 的插入和删除:

(插入元素时): 需要考虑: 内存是否被重新分配

如果容器内存被重新分配, iterators, pointers, references 失效;

(插入删除元素时)

插入删除点之前的 iterator 有效, 插入删除点之后的 iterator 失效;

2. 对于 list 和 forward_list 的插入和删除:

所有的 iterator, pointer 和 reference 都有效

总结: 迭代器失效与否由容器的底层存储结构决定:

顺序存储: vector, deque

重新分配内存空间, 全部失效

it 之前的不失效, it 之后的失效

不连续存储: list, forward_list, map, set

迭代器, 不失效

补充 1: 为什么 vector 的插入操作可能会导致迭代器失效?

当 vector 动态增加大小时, 如果 capacity 不够时, 需要额外申请空间; 将旧空间的数据全部拷贝到新空间中; 释放旧空间。→ 因此, vector 会导致迭代器失效

补充 2: vector、deque、list、map 用 erase (it) 后, 迭代器的变化。

vector 和 deque 是序列式容器, 其内存分别是连续空间和分段连续空间, 删除迭代器 it 后, 伴随着数据的移动和复制, it 前面的迭代器不失效, 但是后面的迭代器都失效了, 此时 it 指向被删除元素的下一个元素。

List 是双向链表存储结构, 地址不连续, 删除迭代器 it 后, 仅仅是指针发生变动, 对其他迭代器不影响, 因此迭代器都不会失效

Map 使用红黑树, 删除迭代器 it, 迭代器依然有效。

面试题 6: list.sort() 和 container.sort() 原理以及性能对比

(1) STL 容器 algorithm 的排序, 支持随机访问的容器: vector, deque, string

原理: 快速排序, 时间复杂度: $O(N \cdot \log N)$

```
1. template <class RandomAccessIterator>
2. void sort ( RandomAccessIterator first, RandomAccessIterator last );
3.
4. template <class RandomAccessIterator, class Compare>
5. void sort ( RandomAccessIterator first, RandomAccessIterator last, Compare comp );
```

STL 容器的 algorithm 的 sort() 函数要求:

1. 随机迭代器, 能用此算法的容器是支持随机访问的容器: vector, deque, string。
2. 要求输入一个范围 [first, last)

3. 第一个版本默认使用 `operator<` 进行比较，默认升序排序；第二个版本可以使用自定义的 `comp` 规则进行比较。

(2) 成员 `list` 排序调用自带的 `list::sort`，不能调用 `STL::sort()` 函数

原理：归并排序，时间复杂度： $O(N \cdot \log N)$

面试题 7：traits 萃取编程（模板，偏特化）

1. 功能：简单的说——判断出正确的参数类型

iterator 通过 traits，实现 `algorithm` 与 `container` 的交互：

`algorithm` 通过向 `iterator` 提问，`iterator` 必须能够回答 `algorithm` 所提出的问题，才能搭配该 `algorithm` 的进行操作。

在此过程中，Traits 起到了回答 `algorithm` 的问题的作用。

那么，问题来了，`algorithm` 到底提出了什么问题呢？其实很简单——`algorithm` 只是在问 `iterator` 五个问题，分别是：

```
template <class Category,
          class T,
          class Distance = ptrdiff_t,
          class Pointer = T*,
          class Reference = T&>

struct iterator
{
    typedef Category iterator_category; 迭代器类型：随机存取/forward/binary/input/output
    typedef T value_type; 迭代器指向的元素的类型
    typedef Distance difference_type; 距离
    typedef Pointer pointer;
    typedef Reference reference; 没使用过，但是必须要写
};
```

再次说明：`iterator` 必须能够回答这五个 `algorithm` 问题，`algorithm` 才能使用算法去处理 `iterator`。

2. 如何使用 traits 指定类型：

方式 1：内嵌类型

```
1. template <class Iterator>
2. struct iterator_traits {
3.     typedef typename Iterator::value_type value_type; // 声明内嵌类型
4.     ...
5. };
6.
7. template <class Iterator>
8. typename iterator_traits<Iterator>::value_type GetValue(Iterator iter)
9. {
10.     return *iter;
11. }
```

代码解读：

typename: 告诉编译器, typename 后面跟的一堆东西是一个数据类型

`iterator_traits<Iterator>::value_type`: 整体表示模板函数的返回类型, 其中 `iterator_traits<Iterator>::` 等价于 `ClassName<TypeName>::`, 表示使用模板类型为 `TypeName`, 类 `ClassName` 的成员属性 `value_type` → 因此返回值类型为 `iterator_traits<Iterator>::value_type`。
说明:

以上这种方法对原生指针不可行, 所以 `iterator_traits` 针对原生指针的一个版本就应运而生。下面是针对 `*Tp` 和 `const *Tp` 的版本, 也称为 `iterator_traits` 版本的偏特化。`iterator_traits` 的偏特化版本解决了原生指针的问题。→→→现在不管迭代器是自定义类模板, 还是原生指针 (`Tp*`, `const Tp*`), `struct iterator_traits` 都能萃取出正确的 `value type` 类型。

方式 2: 原生指针——模板偏特化 (针对于指针类型和 `const` 指针类型)

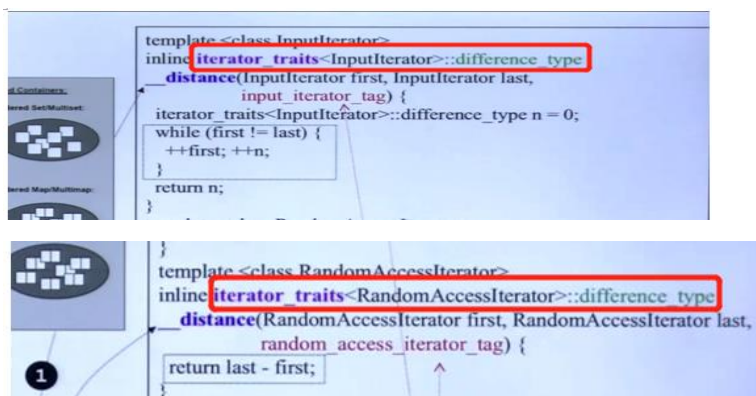
```
1. template <class Tp>
2. struct iterator_traits<Tp*> {
3.     typedef Tp value_type;
4.     ...
5. };
6.
7. template <class Tp>
8. struct iterator_traits<const Tp*> {
9.     typedef Tp value_type;
10.    ...
11.};
```

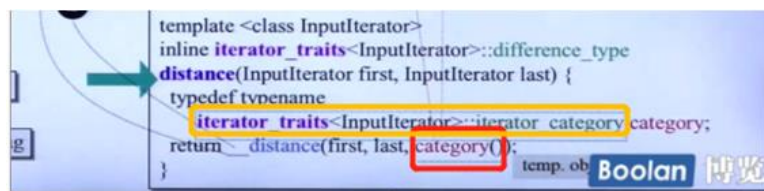
总结:

之所以要萃取 (Traits) 迭代器相关的类型, 就是要将迭代器相关的类型用于声明局部变量、用作函数的返回值等一系列行为。对于原生指针和 `point-to-const` 类型的指针, 采用模板偏特化技术对其进行特殊处理。

1. 要使用 Traits 功能, 则必须自行以内嵌型别定义的方式定义出相应型别。

3. 代码举例: 分析 traits 的使用





上面三个图的代码分析：distance(first,last)函数

(1) distance(first,last) → 调用 __distance(first,last,category()), 其中 category() 的类型是通过 Traits 获得的, 即 `iterator_traits<InputIterator>::iterator_category`

(2) 得到 category 的类型后, 用 category() 创建一个临时对象, 去调用 __distance(first,last,category()), 其中 category() 参数有两种形式: input_iterator_tag 和 random_access_iterator_tag

(3) category() 与 "input_iterator_tag 和 random_access_iterator_tag", 进行类型匹配 (本质上是函数重载), 匹配成功后, 调用匹配的函数, 执行操作

(4) 返回值类型, 同理, 也是通过 Traits 进行获取

`iterator_traits<InputIterator>::difference_type`

4.STL 约定

自定义的迭代器, 必须继承 std::iterator 和 iterator::traits, 才能让它融入 STL 的大家庭中, 无缝的使用各种泛型 algorithm。

面试题 8: 仿函数 (为 algorithm 服务) 类模板

说的通俗点就是在一个类中利用运算符重载重载 "()", 让类拥有函数的使用特性和功能, 这个 class 创建出来的对象, 就是函数对象 (它是一个对象, 但是像一个函数, 因此叫做仿函数)

举例: 一个简单的仿函数

```
1. template<class T>
2. struct A //一个类
3. {
4.     bool operator()(const T& a, const T& b) //重载()运算符
5.     {
6.         return (a == b);
7.     }
8. };
9.
10. int main()
11. {
12.     A<int> aa; //创建一个类对象 aa
13.     cout << aa(1, 2) << endl; //类对象 aa, 调用()函数
14.     return 0;
15. }
```

可适配 (adaptable) 的关键 (继承两个类)

1. 仿函数的相应型别主要用来表现函数参数型别和传回值 **型别**
2. 任何仿函数，只有继承下面其中一个 **class**，才拥有了那些相应 **型别**，也就自动拥有了适配能力，能够被适配器改造。

```
template <class Arg, class Result>
struct unary_function    //一个操作数：如，取负值
{
    typedef Arg argument_type;
    typedef Result result_type;
};
```

```
template <class Arg1, class Arg2, class Result>
struct binary_function   //两个操作数：如，相加，相与，比较大小
{
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

举例说明：List 双向列表的 sort() 排序方法，执行过程种，使用仿函数 less<>()，详细见下：

list类的排序函数sort()

```
List.sort();
```

调用

```
void sort()
{
    // order sequence, using operator<
    sort<less<>()>();
}
// 调用
// 形参是一个仿函数类，创建的临时对象
template<class Pr>
void sort(Pr& _Pred)
{
    // order sequence, using _Pred
    if (2 <= this->_Mysize)
```

其中，仿函数 less 的定义，见下：

```
template<class Ty = void>
struct less    仿函数类less，继承自binary_function
{
    public binary_function< Ty, Ty, bool>
    {
        // functor for operator<
        bool operator()(const Ty& _Left, const Ty& _Right) const
        {
            // apply operator< to operands
            return (_Left < _Right); 并且重写了operator()操作符
        }
    };
```

面试题 9：适配器（3 种）

(1) 容器适配器：deque→stack,queue

本质：Stack 和 queue 使用 deque 作为底层容器，所有的操作都借助 deque 去实现


```
template <class T, class Sequence=deque<T>>
class stack {
...
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c; // 底層容器
public:
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference top() { return c.back(); }
    const_reference top() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};
```

```
template <class T, class Sequence=deque<T>>
class queue {
...
public:
    typedef typename Sequence::value_type value_type;
    typedef typename Sequence::size_type size_type;
    typedef typename Sequence::reference reference;
    typedef typename Sequence::const_reference const_reference;
protected:
    Sequence c; // 底層容器
public:
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference front() { return c.front(); }
    const_reference front() const { return c.front(); }
    reference back() { return c.back(); }
    const_reference back() const { return c.back(); }
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```

(2) 仿函数适配器——bind

新型适配器, bind
Since C++11

www.cplusplus.com/reference/functional/bind/?kw=bind



```
#include <functional> // std::bind

// a function: (also works with function object:
// std::divides<double> my_divide;)
double my_divide(double x, double y)
{ return x / y; }

struct MyPair {
    double a,b;
    double multiply() { return a * b; }
    // member function 其實有個 argument: this
};
```

bind 可以綁定：
- functions
- function objects
- member functions, _1 必須是某個 object 地址。
- data members, _1 必須是某個 object 地址。
一個 function object 相當於 ret。調用 ret 相當於上述 1,2,3，或相當於取出 4。

```
2781 using namespace std::placeholders; // odds visibility of _1, _2, _3,...
2782
2783 // binding functions:
2784 auto fn_five = bind(my_divide,10,2); // returns 10/2
2785 cout << fn_five() << '\n'; // 5
2786
2787 auto fn_half = bind(my_divide,_1,2); // returns x/2
2788 cout << fn_half(10) << '\n'; // 5
2789
2790 auto fn_invert = bind(my_divide,_2,_1); // returns y/x
2791 cout << fn_invert(10,2) << '\n'; // 0.2
2792
2793 auto fn_rounding = bind(int, my_divide,_1,_2); // returns int(x/y)
2794 cout << fn_rounding(10,3) << '\n'; // 3
2795
2796 // binding members:
2797 MyPair ten_two {10,2};
2798
2799 // member function 其實有個 argument: this
2800 auto bound_memfn = bind(&MyPair::multiply, _1); // returns x.multiply()
2801 cout << bound_memfn(ten_two) << '\n'; // 20
2802
2803 auto bound_memdata = bind(&MyPair::a, ten_two); // returns ten_two.a
2804 cout << bound_memdata() << '\n'; // 10
2805
2806 auto bound_memdata2 = bind(&MyPair::b, _1); // returns x.a b
2807 cout << bound_memdata2(ten_two) << '\n'; // 2
2808
2809 vector<int> v {15,37,94,50,73,58,28,98};
2810 int n = count_if(v.cbegin(), v.cend(), not1(bind2nd(less<int>(),50)));
2811 cout << "n = " << n << endl; //5
2812
2813 auto fn_ = bind(less<int>(),_1,50);
2814 cout << count_if(v.cbegin(), v.cend(), fn_) << endl;
2815 cout << count_if(v.begin(), v.end(), bind(less<int>(), 1,50)) << endl; //5
```

(3) 迭代器适配器

迭代器适配器：
reverse_iterator

```
reverse_iterator
rbegin()
{ return reverse_iterator(end()); }

reverse_iterator
rend()
{ return reverse_iterator(begin()); }
```

面试题 10: Tuple

tuple 元之組合, 數之組合

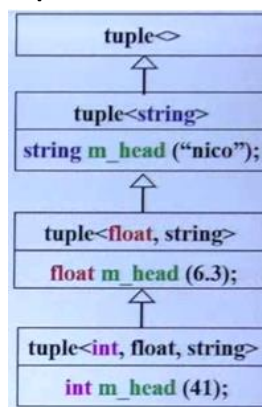
G4.8 節錄並簡化

```
template<typename... Values> class tuple;
template<> class tuple<> { };

template<typename Head, typename... Tail>
class tuple<Head, Tail...>
: private tuple<Tail...> 递归继承
{
    typedef tuple<Tail...> inherited;
public:
    tuple() { }
    tuple(Head v, Tail... vtail)
    : m_head(v), inherited(vtail...) { }
    // 呼叫 base ctor 並予參數 (不是創建 temp object)
    // 注意這是 initialization list

    typename Head::type head() { return m_head; }
    inherited& tail() { return *this; }
    // return 後轉型為 inherited, 獲得的是
protected:
    Head m_head;
};
```

Tuple 实现的关键: 递归继承, 见下图



面试题 11: STL 内存分配

STL 内存分配

<http://www.cnblogs.com/lang5230/p/5556611.html>

STL 相关: STL 中的内存管理(allocator)的原理, 以及如何让它线程安全。

<http://yaocoder.blog.51cto.com/2668309/1208465/>