

2018年春

程序设计实习 (I): C++程序设计

第九讲 标准模板库 1

刘家瑛

liujiaying@pku.edu.cn



课前多吼歪

- 本周上机正常, 4月21日停一次
 - 特殊情况请邮件助教请假
 - 不要在群里请假
- 习题分享(下周三继续招募)
- 关于尽力多谈一点



上节课知识点回顾

- 函数模板
 - 函数重载 vs. 函数模板
 - 类型参数
 - 函数和模板的匹配顺序
- 类模板
 - 类模板 & 模板类
 - 使用类模板声明对象: 同一个类模板的两个模板类是不兼容
 - 类模板的非类型参数
 - 类模板与继承: 4种情况
 - 类模板与友元函数: 4种情况
 - 类模板与静态变量: 不同的模板类不能共享静态变量



上节课知识点回顾

- 函数模板
 - 函数重载 vs. 函数模板
 - 类型参数
 - 函数和模板的匹配顺序
- 类模板
 - 类模板 & 模板类
 - 使用类模板声明对象: 同一个类模板的两个模板类是不兼容
 - 类模板的非类型参数
 - 类模板与继承: 4种情况
 - 类模板与友元函数: 4种情况
 - 类模板与静态变量: 不同的模板类不能共享静态变量



上节课知识点回顾

函数模板

template <class 类型参数1, class 类型参数2, ... >

返回值类型 模板名 (形参表)

{ 函数体 }

- 1) 先找一个参数完全匹配的函数
- 2) 再找一个参数完全匹配的模板
- 3) 在没有二义性的前提下, 再找一个参数经过自动转换后能够匹配的函数



上节课知识点回顾

- 函数模板
 - 函数重载 vs. 函数模板
 - 类型参数
 - 函数和模板的匹配顺序
- 类模板
 - 类模板 & 模板类
 - 使用类模板声明对象: 同一个类模板的两个模板类是不兼容
 - 类模板的非类型参数
 - 类模板与继承: 4种情况
 - 类模板与友元函数: 4种情况
 - 类模板与静态变量: 不同的模板类不能共享静态变量



上节课知识点回顾

类模板

template <类型参数表>

class 类模板名

{ 成员函数和成员变量 };

• 继承

类模板派生出类模板; 模板类派生出类模板;

普通类派生出类模板; 模板类派生出普通类;

• 友元

函数/类/类的成员函数作为类模板的友元;

函数模板作为类模板的友元; 函数模板作为类的友元;

类模板作为类模板的友元;



上节课知识点复习

以下说法错误的是：

- A. 编译器根据调用模板时的实参, 从函数模板实例化出模板函数
- B. 函数模板中, 同一个类型参数, 实例化时可以用不同类型替代
- C. 函数模板的返回值可以是确定的类型, 也可以是由类型参数表示的非确定类型
- D. 从一个函数模板可以实例化出多个模板函数



上节课知识点复习

下面那种派生是不可行的？

- A. 从类模版派生出类模版
- B. 从模版类派生出类模版
- C. 从类模版派生出普通类
- D. 从普通类派生出类模版

正确：

从模板类派生出
普通类



上节课知识点复习

string类的at成员函数和[]成员函数的区别是：

- A. at运行速度比 [] 更快
- B. at的返回值是指针, []的返回值是引用
- C. []比at更符合使用习惯, 除此之外没有差别
- D. at要检查下标的范围, []不作此检查



主要内容

- 基本概述与概念
- 容器概述
 - 顺序容器介绍
 - 关联容器介绍
 - 容器适配器介绍
- 迭代器
- 算法概述
- 顺序容器
 - **vector/list/deque**



标准模板库 STL

- C++ 语言的核心优势之一 —— **软件的重用**
- C++中有两个方面体现重用:
 1. **面向对象**的思想: 继承, 多态, 标准类库
 2. **泛型程序设计** (Generic Programming) 的思想:
模板机制, 标准模板库 STL



标准模板库 STL

泛型程序设计:

使用**模板**的程序设计方法

将一些常用的**数据结构** (比如链表, 数组, 二叉树)

和**算法** (比如排序, 查找) 写成模板

- 数据结构里放的是什么对象/算法针对什么对象

→ 都不必重新实现数据结构, 重新编写算法



标准模板库 STL

标准模板库 (Standard Template Library)

- 一些常用数据结构和算法的模板的集合
- 主要由 Alex Stepanov 开发, 98年被添加进C++标准

有了STL, 不必再写大多的标准数据结构和算法,
并且可获得非常高的性能



STL中有几个基本的概念:

- **容器**: 可容纳各种数据类型的数据结构
- **迭代器**: 可依次存取容器中元素的工具
 - 普通的C++指针就是一种迭代器
- **算法**: 用来操作容器中元素的函数模板



例如：STL用**sort()**来对一个**vector**中的数据进行排序

用**find()**来搜索一个**list**中的对象

函数本身与其操作的数据的结构和类型无关

→可从简单数组到高度复杂容器的任何数据结构上使用

***e.g.* int array[100];**

这个数组就是个容器, 而 `int *` 类型的指针变量就可以作为迭代器, 可以为这个容器编写一个排序的算法



1 容器概述

- 可以用于存放各种类型的数据 (基本类型的变量, 对象等) 的数据结构
- 容器分为三大类:
 - 1) 顺序容器/序列容器
vector, deque, list
 - 2) 关联容器/有序容器
set, multiset, map, multimap
以上两种容器称为第一类容器
 - 3) 容器适配器
stack, queue, priority_queue



- 对象被插入容器中时, 被插入的是**对象的一个复制品**
- 许多算法, 比如排序, 查找, 要求对容器中的元素进行比较, 所以放入容器的**对象所属的类**, 还应该**实现 == 和 < 运算符**



1.1 顺序容器简介

1) vector 头文件 <vector>

实际上就是个**动态数组**. 随机存取任何元素都能在常数时间完成. 在**尾端增删元素**具有较佳的性能

2) deque 头文件 <deque>

也是个**动态数组**, 随机存取任何元素都能在常数时间完成 (但次于vector). 在**两端增删元素**具有较佳的性能

3) list 头文件 <list>

双向链表, 在任何位置增删元素都能在常数时间完成. **不支持随机存取**

上述三种容器称为顺序容器, 是因为**元素的插入位置同元素的值无关**



1.2 关联容器简介

关联式容器内的元素是**排序**的, 插入任何元素, 都按相应的排序准则来确定其位置. 关联式容器的特点是在查找时具有非常好的性能

1) set/multiset: 头文件 <set>

集合, set不允许相同元素, multiset中允许存在相同的元素

2) map/multimap: 头文件 <map>

映射, map与set的不同在于map中存放的是成对的**key/value**

并根据key对元素进行排序, 可快速地根据key来检索元素

map同multimap的不同在于是否允许相同key的元素

上述四种容器通常以平衡二叉树方式实现

插入和检索的时间都是 $O(\log N)$



1.3 容器适配器简介

1) **stack**: 头文件 `<stack>`

栈. 是项的有限序列, 并满足序列中被删除、检索和修改的项只能是最近插入序列的项. 即按照**后进先出**的原则

2) **queue**: 头文件 `<queue>`

队列. 插入只可以在尾部进行, 删除、检索和修改只允许从头部进行. 按照**先进先出**的原则

3) **priority_queue**: 头文件 `<queue>`

优先级队列. 最高优先级元素总是第一个出列



1.4 容器的共有成员函数

1) 所有标准库容器共有的成员函数:

- 相当于按词典顺序比较两个容器的运算符:
=, <, <=, >, >=, ==, !=
- **empty**: 判断容器中是否有元素
- **max_size**: 容器中最多能装多少元素
- **size**: 容器中元素个数
- **swap**: 交换两个容器的内容



比较两个容器的例子：

```
#include <vector>
#include <iostream>
using namespace std;
class A {
    private:
        int n;
    public:
        friend bool operator < (const A &, const A &);
        A(int n_) { n = n_; }
};
bool operator < (const A & o1, const A & o2) {
    return o1.n < o2.n;
}
```



```
int main(){  
    vector<A> v1;  
    vector<A> v2;  
    v1.push_back (A(5));  
    v1.push_back (A(1));  
    v2.push_back (A(1));  
    v2.push_back (A(2));  
    v2.push_back (A(3));  
    cout << (v1 < v2);  
    return 0;  
}
```

输出：

0



2) 只在第一类容器中的函数:

- **begin** 返回指向容器中**第一个元素**的迭代器
- **end** 返回指向容器中**最后一个元素后面**的位置的迭代器
- **rbegin** 返回指向容器中**最后一个元素**的迭代器
- **rend** 返回指向容器中**第一个元素前面**的位置的迭代器
- **erase** 从容器中删除一个或几个元素
- **clear** 从容器中删除所有元素



2 迭代器

- 用于指向**第一类容器**中的元素, 有const 和非 const两种
- 通过迭代器可以读取它指向的元素
通过非const迭代器还能修改其指向的元素
迭代器用法和指针类似
- 定义一个容器类的迭代器的方法可以是:
容器类名::iterator 变量名;
或:
容器类名::const_iterator 变量名;
- 访问一个迭代器指向的元素:
*** 迭代器变量名**



- 迭代器可以**执行 ++** 操作，以指向容器中的下一个元素
- 如果迭代器到达了容器中的最后一个元素的后面，则迭代器变成past-the-end值
 - 使用一个past-the-end值的迭代器来访问对象是非法的
 - 类似用NULL或未初始化的指针一样



例如：

```
#include <vector>
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> v; //一个存放int元素的向量, 一开始里面没有元素
```

```
    v.push_back(1);
```

```
    v.push_back(2);
```

```
    v.push_back(3);
```

```
    v.push_back(4);
```

```
    vector<int>::const_iterator i; //常量迭代器
```

```
    for( i = v.begin(); i != v.end(); i ++ )
```

```
        cout << * i << ", ";
```

```
    cout << endl;
```



```
vector<int>::reverse_iterator r; //反向迭代器
```

```
for( r = v.rbegin(); r != v.rend(); r++ )
```

```
    cout << * r << ", ";
```

```
cout << endl;
```

```
vector<int>::iterator j; //非常量迭代器
```

```
for( j = v.begin(); j != v.end(); j ++ )
```

```
    * j = 100;
```

```
for( i = v.begin(); i != v.end(); i++ )
```

```
    cout << * i << ", ";
```

```
}
```

输出结果:

1, 2, 3, 4,

4, 3, 2, 1,

100, 100, 100, 100,



容器与迭代器

- 不同容器上支持的迭代器功能强弱有所不同
- 容器的迭代器的功能强弱, 决定了该容器**是否支持**

STL中的某种算法

- 只有**第一类容器**能用迭代器遍历
- 排序算法需要通过随机迭代器来访问容器中的元素,
那么有的容器就不支持排序算法



STL中的迭代器

STL 中的迭代器按功能由弱到强分为5种：

1. 输入：Input iterators 提供对数据的只读访问
1. 输出：Output iterators 提供对数据的只写访问
2. 正向：Forward iterators 提供读写操作, 并能一次一个地向前推进迭代器
3. 双向：Bidirectional iterators 提供读写操作, 并能一次一个地向前和向后移动
4. 随机访问：Random access iterators 提供读写操作, 并能在数据中随机移动

编号大的迭代器拥有编号小的迭代器的所有功能, 能当作编号小的迭代器使用



不同迭代器所能进行的操作(功能):

- 所有迭代器: $++p$, $p++$
- 输入迭代器: $*p$, $p = p1$, $p == p1$, $p != p1$
- 输出迭代器: $*p$, $p = p1$
- 正向迭代器: 上面全部
- 双向迭代器: 上面全部, $--p$, $p--$,
- 随机访问迭代器: 上面全部, 以及:
 - 移动*i*个单元:** $p += i$, $p -= i$, $p + i$, $p - i$
 - 大于/小于比较:** $p < p1$, $p \leq p1$, $p > p1$, $p \geq p1$
 - 数组下标*p[i]*:** p 后面的第*i*个元素的引用



容器

vector

deque

list

set/multiset

map/multimap

stack

queue

priority_queue

迭代器类别

随机

随机

双向

双向

双向

不支持迭代器

不支持迭代器

不支持迭代器



例如, vector的迭代器是**随机迭代器**, 所以遍历 vector 可以有以下几种做法:

```
vector<int> v(100);
```

```
int i;
```

```
for(i = 0; i < v.size(); i ++)
```

```
    cout << v[i];
```

```
vector<int>::const_iterator ii;
```

```
for( ii = v.begin(); ii != v.end (); ii ++ )
```

```
    cout << * ii;
```

```
for( ii = v.begin(); ii < v.end (); ii ++ )
```

```
    cout << * ii;
```



//间隔一个输出：

```
ii = v.begin();
```

```
while( ii < v.end()) {
```

```
    cout << * ii;
```

```
    ii = ii + 2;
```

```
}
```



而 list 的迭代器是**双向迭代器**, 所以以下代码可以:

```
list<int> v;
```

```
list<int>::const_iterator ii;
```

```
for( ii = v.begin(); ii != v.end (); ii ++ )
```

```
    cout << * ii;
```

以下代码则不行:

```
for( ii = v.begin(); ii < v.end (); ii ++ )
```

```
    cout << * ii;
```

//双向迭代器不支持 <

```
for(int i = 0; i < v.size(); i ++)
```

```
    cout << v[i]; //双向迭代器不支持 []
```



3 算法简介

STL中提供能在各种容器中通用的算法, 比如插入/删除/查找/排序等. 大约有70种标准算法

- 算法就是一个个函数模板
- 算法通过迭代器来操纵容器中的元素
- 许多算法需要两个参数, 一个是起始元素的迭代器,
- 一个是终止元素的后面一个元素的迭代器
 - 排序和查找
- 有的算法返回一个迭代器. 比如 `find()` 算法, 在容器中查找一个元素, 并返回一个指向该元素的迭代器
- 算法可以处理容器, 也可以处理C语言的数组



1) 变化序列算法:

copy, remove, fill, replace, random_shuffle, swap, ...

会改变容器

2) 非变化序列算法:

adjacent-find, equal, mismatch, find, count, search,
count_if, for_each, search_n

以上函数模板都在`<algorithm>`中定义
还有其他算法, 比如`<numeric>`中的算法



算法示例: **find()**

```
template<class InIt, class T>
```

```
InIt find(InIt first, InIt last, const T& val);
```

- first 和 last 这两个参数都是容器的迭代器, 它们给出了容器中的查找区间起点和终点
 - 这个区间是个左闭右开的区间, 即区间的起点是位于查找范围之中的, 而终点不是
- val参数是要查找的元素的值
- 函数返回值是一个迭代器
 - 如果找到, 则该迭代器指向被找到的元素
 - 如果找不到, 则该迭代器指向查找区间终点



```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main() {
    int array[10] = {10, 20, 30, 40};
    vector<int> v;
    v.push_back(1);    v.push_back(2);
    v.push_back(3);    v.push_back(4);
    vector<int>::iterator p;
    p = find(v.begin(), v.end(), 3);
    if( p != v.end())
        cout << * p << endl;
```



```

p = find(v.begin(), v.end(), 9);
if( p == v.end())
    cout << "not found " << endl;
p = find(v.begin()+1, v.end()-2, 1); //查找区间[2, 3)
if( p != v.end())
    cout << * p << endl;
int * pp = find(array, array+4, 20);
cout << * pp << endl;
}

```

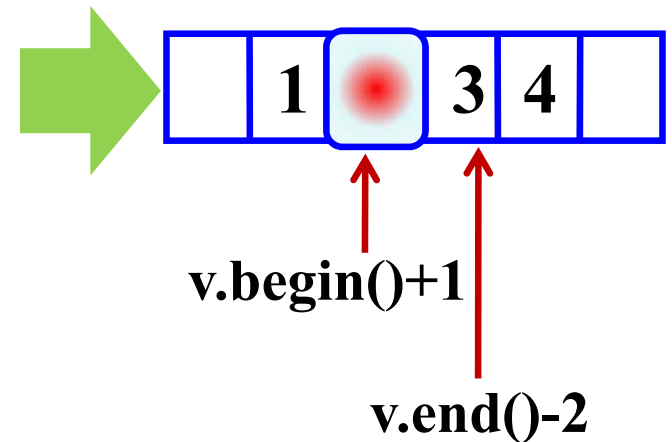
输出:

3

not found

3

20



STL中 "大/小/相等" 的概念

- STL中, 缺省的情况下, 比较大小是用 "<" 运算符进行的, 和 ">" 运算符无关
- 使用STL时, 在缺省的情况下, 以下三个说法等价:
 - x 比y 小
 - 表达式 "x<y" 为真
 - y 比x 大
- 与 ">" 无关, ">" 可以没定义



STL中“大/小/相等”的概念

在STL中“x和y相等”往往不等价于“x==y为真”

- 对于在**未排序**的区间上进行的算法, 比如顺序查找find, 查找过程中比较两个元素是否相等, 用的是**==**运算符

-
- 对于在**排好序**的区间上进行查找, 合并等操作的算法 (如折半查找算法binary_search, 关联容器自身的成员函数find)

- “x和y相等” \Leftrightarrow “x<y和y<x同时为假” 等价的
- 与**==**运算符无关



STL 中 "相等" 概念演示

```
#include <iostream>
#include <algorithm>
using namespace std;
class A {
    int v;
public:
    A(int n):v(n) { }
    bool operator < ( const A & a2) const {
        cout << v << "<" << a2.v << "?" << endl;
        return false;
    }
    bool operator ==(const A & a2) const {
        cout << v << "==" << a2.v << "?" << endl;
        return v == a2.v;
    }
};
```



STL中"相等"概念演示

```
int main()
{
    A a [] = { A(1), A(2), A(3), A(4), A(5) };
    cout << binary_search(a, a+4, A(9)); //折半查找
    return 0;
}
```

输出结果:

3<9?

2<9?

1<9?

9<1?

1



STL概述小结

- **STL: 容器+迭代器+算法**
- **容器**
 - 3类容器: 顺序容器, 关联容器, 容器适配器
 - 容器的共有成员函数:
只在第一类容器中的函数(begin/end, rbegin/rend, erase/clear)
- **迭代器**
 - 5种迭代器及其所支持的操作(功能)
 - 容器所支持的迭代器类别
- **算法**
 - **find()函数**: 搜索区间是**左闭右开**的区间
若找不到则返回的迭代器指向查找区间终点
 - **copy()函数**



4 顺序容器

除前述共同操作外, 顺序容器还有以下共同操作:

- **front()**: 返回容器中第一个元素的引用
- **back()**: 返回容器中最后一个元素的引用
- **push_back()**: 在容器末尾增加新元素
- **pop_back()**: 删除容器末尾的元素

比如, 查 `list::front` 的help, 得到的定义是:

- `reference front();`
- `const_reference front() const;`

list有两个front函数



reference 和 const_reference 是typedef的类型
对于 list<double> ,

- **list<double>::reference** 实际上就是 **double &**
 - **list<double>::const_reference** 实际上就是 **const double &**
- 对于 list<int> ,
- **list<int>::reference** 实际上就是 **int &**
 - **list<int>::const_reference** 实际上就是 **const int &**



4.1 vector

- 支持随机访问迭代器, 所有STL算法都能对vector操作
- 随机访问时间为常数
- 在尾部添加速度很快, 在中间插入慢
- 实际上就是**动态数组**



例1:

```
int main() {  
    int i;  
    int a[5] = {1, 2, 3, 4, 5};  
    vector<int> v(5);  
    cout << v.end() - v.begin() << endl;  
    for( i = 0; i < v.size(); i ++ ) v[i] = i;  
    v.at(4) = 100;  
    for( i = 0; i < v.size(); i ++ )  
        cout << v[i] << ", " ;  
    cout << endl;  
    vector<int> v2(a, a+5); //构造函数  
    v2.insert( v2.begin() + 2, 13 ); //在begin()+2位置插入 13  
    for( i = 0; i < v2.size(); i ++ )  
        cout << v2[i] << ", " ;  
}
```

输出:

5

0, 1, 2, 3, 100,

1, 2, 13, 3, 4, 5,




```
#include <vector>
#include <algorithm>
#include <iostream>
#include <stdexcept>
#include <iterator>
using namespace std;
int main() {
    const int SIZE = 5;
    int a[SIZE] = {1, 2, 3, 4, 5 };
    vector<int> v (a, a+5); //构造函数
    try {
        v.at(100) = 7;
    } //提供了一种方法来处理可能发生在给定代码块中的
        //某些或全部错误, 同时仍保持代码的运行
    catch( out_of_range e) {
        cout << e.what() << endl;
    } //at()会做范围检查, 若超出范围会抛出out_of_range异常
    cout << v.front() << “, ” << v.back() << endl;
```

```
v.erase(v.begin());  
ostream_iterator<int> output(cout, "*");  
copy (v.begin(), v.end(), output);  
v.erase(v.begin(), v.end()); //等效于 v.clear();  
if( v.empty ()  
    cout << "empty" << endl;  
v.insert (v.begin(), a, a+SIZE);  
copy (v.begin(), v.end(), output);  
return 0;  
}
```

输出：

```
invalid vector<T> subscript //(DEV输出)vector::_M_range_check  
1, 5  
2*3*4*5*empty  
1*2*3*4*5*
```

copy 函数

- `ostream_iterator<int> output (cout , "*");`
 - 定义了一个 `ostream_iterator` 对象,
 - 可以通过 `cout` 输出以 * 分隔的一个个整数
- `copy (v.begin(), v.end(), output);`
 - 导致 `v` 的内容在通过 `output` 上输出
 - `first` 和 `last` 的类型是 `vector<int>::const_iterator`
 - `output` 的类型是 `ostream_iterator<int>`



copy 函数

copy 函数模板(算法):

template<class InIt, class OutIt>

OutIt copy(InIt first, InIt last, OutIt x);

- 本函数对在区间 $[0, \text{last} - \text{first})$ 中的每个 N 执行一次

$*(x+N) = *(\text{first} + N)$, 返回 $x+N$



copy的源代码

```
template<class _II, class _OI>
inline _OI copy(_II _F, _II _L, _OI _X)
{
    for (; _F != _L; ++_X, ++_F)
        *_X = *_F;
    return (_X);
}
```



iostream迭代器

- 模板类 **ostream_iterator** 描述一个输出迭代器对象
→ 该对象使用提取运算符 << 将连续的元素写入输出流
- 模板类 **istream_iterator** 描述一个输入迭代器对象
 - **istream_iterator**: 读取输入流, 支持比较(==, !=), 解引用(*, ->), 自增(++)
 - **ostream_iterator**: 写输出流, 支持解引用(*, ->), 自增(++)



iostream迭代器

例:

```
ostream_iterator<string> out_iter(cout, "\\n");
```

```
istream_iterator<string> in_iter(cin), eof; //eof为空迭代器
```

```
while (in_iter != eof)
```

```
    *out_iter++ = *in_iter++; //先执行++, 返回原值副本  
                             //再执行解引用*来取值
```



关于 ostream_iterator, istream_iterator的例子

```
int main() {  
    istream_iterator<int> inputInt(cin);  
    int n1, n2;  
    n1 = * inputInt; //读入 n1  
    inputInt++;  
    n2 = * inputInt; //读入 n2  
    cout << n1 << ", " << n2 << endl;  
    ostream_iterator<int> outputInt(cout);  
    * outputInt = n1 + n2;  
    cout << endl;  
    int a[5] = {1, 2, 3, 4, 5};  
    copy(a, a+5, outputInt); //输出整个数组  
    return 0;  
}
```



程序运行后输入 78 90敲回车, 则输出结果为:

78, 90

168

12345



```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;
int main(){
    int a[4] = { 1, 2, 3, 4 };
    My_ostream_iterator<int> oit(cout, "*");
    copy( a, a+4, oit); //输出 1*2*3*4*
    ofstream oFile("test.txt", ios::out);
    My_ostream_iterator<int> oitf(oFile, "*");
    copy(a, a+4, oitf); //向test.txt文件中写入 1*2*3*4*
    oFile.close();
    return 0;
} // 如何编写 My_ostream_iterator?
```



copy 的源代码:

```
template<class _II, class _OI>
inline _OI copy(_II _F, _II _L, _OI _X){
    for (; _F != _L; ++_X, ++_F)
        *_X = *_F;
    return (_X);
}
```

上面程序中调用语句 “copy(a, a+4, oit)” 实例化后得到copy如下:

```
My_ostream_iterator<int> copy(int * _F, int * _L,
                               My_Ostream_iterator<int> _X){
    for (; _F != _L; ++_X, ++_F)
        *_X = *_F;
    return (_X);
}
```



My_ostream_iterator类应该重载“++”和“*”运算符,
“=”也应该被重载

template<class T>

class My_ostream_iterator{
 private:

 string sep; //分隔符
 ostream & os;

public:

My_ostream_iterator(ostream & o, string s):sep(s), os(o){ }

void operator ++() { }; // ++只需要有定义即可

My_ostream_iterator & operator * ()

 { return * this; }

My_ostream_iterator & operator = (const T & val)

 { os << val << sep; return * this; }

};



4.2 list 容器

- 在任何位置插入/删除都是**常数时间**, **不支持随机存取**
- 除了具有所有顺序容器都有的成员函数以外, 还支持8个成员函数:
 - **push_front**: 在**前面插入**
 - **pop_front**: **删除前面**的元素
 - **sort**: 排序 (list 不支持 STL 的算法 sort)
 - **remove**: 删除和指定值相等的所有元素
 - **unique**: 删除所有和前一个元素相同的元素
 - **merge**: 合并两个链表, 并清空被合并的那个
 - **reverse**: 颠倒链表
 - **splice**: 在指定位置前面插入另一链表中的一个或多个元素, 并在另一链表中删除被插入的元素



list容器之sort函数

- list容器的迭代器不支持完全随机访问, 所以不能用标准库中sort函数对它进行排序
- list自己的sort成员函数

`list<T> classname`

`classname.sort(compare);` //compare函数可以自己定义

`classname.sort();` //无参数版本, 按<排序

- 与其他顺序容器不同, list容器只能使用双向迭代器→
不支持大于/小于比较运算符, []运算符和随机移动
(即类似 "list的迭代器+2" 的操作)



```
#include <list>
```

```
#include <iostream>
```

```
#include <algorithm>
```

```
using namespace std;
```

```
class A { //定义类A, 并以友元重载<, ==和<<
```

```
private:
```

```
    int n;
```

```
public:
```

```
    A( int n_ ) { n = n_; }
```

```
    friend bool operator<( const A & a1, const A & a2 );
```

```
    friend bool operator==( const A & a1, const A & a2 );
```

```
    friend ostream & operator <<( ostream & o, const A & a );
```

```
};
```



```
bool operator<( const A & a1, const A & a2 ) {  
    return a1.n < a2.n;  
}  
  
bool operator==( const A & a1, const A & a2 ) {  
    return a1.n == a2.n;  
}  
  
ostream & operator <<( ostream & o, const A & a ) {  
    o << a.n;  
    return o;  
}
```



//定义函数模板PrintList, 打印列表中的对象

```
template <class T>
```

```
void PrintList(const list<T> & lst) {
```

```
    int tmp = lst.size();
```

```
    if( tmp > 0 ) {
```

```
        typename list<T>::const_iterator i;
```

```
        for( i = lst.begin(); i != lst.end(); i ++ )
```

```
            cout << * i << ", ";
```

```
    }
```

```
}
```

//与其他顺序容器不同, list容器只能使用双向迭代器,
因此不支持大于/小于比较运算符, []运算符和随机移动

// typename用来说明 list<T>::const_iterator是个类型

//在VS中不写也可以



```
int main() {  
    list<A> lst1, lst2;  
    lst1.push_back(1); lst1.push_back(3); lst1.push_back(2);  
    lst1.push_back(4); lst1.push_back(2);  
    lst2.push_back(10); lst2.push_front(20);  
    lst2.push_back(30); lst2.push_back(30);  
    lst2.push_back(30); lst2.push_front(40);  
    lst2.push_back(40);  
    cout << "1) "; PrintList( lst1); cout << endl;  
    cout << "2) "; PrintList( lst2); cout << endl;  
    lst2.sort(); //list容器的sort函数  
    cout << "3) "; PrintList( lst2); cout << endl;  
}
```

1) 1, 3, 2, 4, 2,

2) 40, 20, 10, 30, 30, 30, 40,

3) 10, 20, 30, 30, 30, 40, 40,



```
lst2.pop_front();
```

```
cout << "4) "; PrintList( lst2); cout << endl;
```

```
lst1.remove(2); //删除所有和A(2)相等的元素
```

```
cout << "5) "; PrintList( lst1); cout << endl;
```

```
lst2.unique(); //删除所有和前一个元素相等的元素
```

```
cout << "6) "; PrintList( lst2); cout << endl;
```

```
lst1.merge( lst2); //合并 lst2到lst1并清空lst2
```

```
cout << "7) "; PrintList( lst1); cout << endl;
```

```
cout << "8) "; PrintList( lst2); cout << endl;
```

```
lst1.reverse();
```

```
cout << "9) "; PrintList( lst1); cout << endl;
```

4) 20, 30, 30, 30, 40, 40,

5) 1, 3, 4,

6) 20, 30, 40,

7) 1, 3, 4, 20, 30, 40,

8)

9) 40, 30, 20, 4, 3, 1,



```

lst2.push_back (100); lst2.push_back (200);
lst2.push_back (300); lst2.push_back (400);
list<A>::iterator p1, p2, p3;
p1 = find(lst1.begin(), lst1.end(), 3);
p2 = find(lst2.begin(), lst2.end(), 200);
p3 = find(lst2.begin(), lst2.end(), 400);
lst1.splice(p1, lst2, p2, p3); //将[p2, p3)插入p1之前,
                                //并从lst2中删除[p2, p3)

cout << "11) "; PrintList(lst1); cout << endl;
cout << "12) "; PrintList(lst2); cout << endl;
return 0;
}

```

11) 40, 30, 20, 4, 200, 300, 3, 1,
 12) 100, 400,



输出：

- 1) 1, 3, 2, 4, 2,
- 2) 40, 20, 10, 30, 30, 30, 40,
- 3) 10, 20, 30, 30, 30, 40, 40,
- 4) 20, 30, 30, 30, 40, 40,
- 5) 1, 3, 4,
- 6) 20, 30, 40,
- 7) 1, 3, 4, 20, 30, 40,
- 8)
- 9) 40, 30, 20, 4, 3, 1,
- 11) 40, 30, 20, 4, 200, 300, 3, 1,
- 12) 100, 400,

