2024年春

程序设计实习：C++程序设计

第十讲　标准模板库STL-1

贾川民
北京大学

课前提醒

- 清明假期：4.7上周五的课，本周上机暂停一次
- 魔兽大作业的相关文件已在教学网发布

魔兽1是为了让大家熟悉使用面向对象的思想进行编程，最好不要使用一个大函数来处理，后面的魔兽世界会更加复杂。给出的参考代码思路非常清晰，建议大家阅读一下，对后面的编程会有很大帮助。附件是郭炜老师参考标程。
另外在poj上做题的时候一定要看清题目的输出要求，能减少许多debug时的工作量。

建议大家一定看一看。参考代码的思路非常清晰。
魔兽2与魔兽1相比其实就是每次生产warrior时多输出了一句话，参考程序使用了派生和多态。
清晰的思路有助于编程，尤其是对于魔兽终极版来说...
另外看清题目也非常重要，有的同学没看到武器分配n%3这一句，每次给武士分配的都是同一种武器
还有输出时也要细心，有的同学输出时It's 没有加'，有的在设定小数位数时没写正确，导致错误
认真审题，小心输出，磨刀不误砍柴工。

课前提醒

- BMP文件旋转作业发布
- 课程安排：4月3/7日 STL内容（1、2）
- 课程安排：4月10日 C++11及高阶
- 课程安排：4月17日，QT内容介绍
- 期中考试初步计划：4月27日下午

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** (例如 链表, 数组, 二叉树)

和**算法** (例如 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

标准模板库 STL

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**

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

1 容器概述

- 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，例如 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>

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(logN)

1.3 容器适配器简介

1) stack: 头文件 <stack>

栈. 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 迷代器

- 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**

2 迷代器

- 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：

```
#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 <= p1, p > p1, p>= p1

  数组下标p[i]: p 后面的第i个元素的引用

不同迭代器所能进行的操作(功能):

容器	迭代器类别
vector	随机
deque	随机
list	双向
set/multiset	双向
map/multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	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;
}
```

```
//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(inti = 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**, 例如插入/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**. 例如 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**

3 算法简介

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, 例如 **<numeric>** 中的算法

3 算法简介

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;
}
```

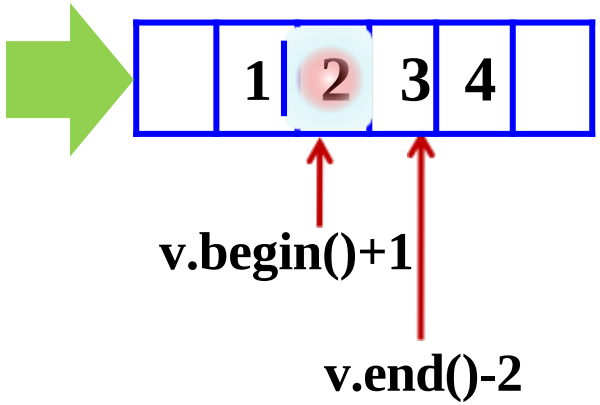
输出:

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, 例如顺序查找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" <= => "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 {
        intv;
    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
```

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

例：查 list::front 的help，得到的定义是：

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

list有**两个front函数**

4 顺序容器

reference 和 const_reference 是typedef的类型

对于 list<double> ,

- **list<double>::refrence** 实际上就是 **double &**
- **list<double>::const_refreence** 实际上就是 **const double &**

对于 list<int> ,

- **list<int>::refrence** 实际上就是 **int &**
- **list<int>::const_refreence** 实际上就是 **constint &**

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 output**
  - 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, last - first)**中的每个N执行一次 ***(x+N) = * (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迷代器

- 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**: 写入输出流，支持解引用(*，->)，自增(++)

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:
          stringsep;      //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) {
    inttmp = 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,

4.3 deque 容器

- 所有适用于 vector的操作都适用于 deque
- 比vector的优点: **头部删除/添加元素**性能也很好
- deque还包含以下操作:
  - **push_front:** 将元素插入到前面
  - **pop_front:** 删除最前面的元素

顺序容器: 小 结

- **vector:** 强调通过**随机访问**进行快速访问
  - 插入/删除：非尾处O(n)或结尾处O(1)
- **list:** 强调元素的**快速插入和删除**, 不支持**随机访问**迭代器
  - 插入/删除为O(1)
- **deque:** 类似vector容器, 但强调在**两端**处的**快速插入和删除**
  - 均为O(1)

5 函数对象

- 一个类**重载了( )为成员函数** → 该类为**函数对象类**
- 这个类的对象 → **函数对象**
- 看上去像函数调用，实际上也执行了函数调用

5 函数对象

- **函数对象 - 目的**
  - 为了**STL算法** 可复用, 其中的子操作应该是参数化的
  - 例如： sort的排序原则 (顺序/ 逆序)
  - **函数对象** 就是用来描述这些**子操作**的**对象**

5 函数对象

```
class CMyAverage {
    public:
        double operator() (int a1,  int a2,  int a3) {
       //重载 ( ) 运算符
            return (double)(a1 + a2 + a3) / 3;
        }
}; //重载 ( ) 运算符时, 参数可以是任意多个
 CMyAverage Average;    //函数对象
 cout << Average(3, 2, 3); // Average.operator(3, 2, 3)  用起来
                           // 看上去像函数调用
                           // 输出 2.66667
```

函数对象的应用

STL里有以下模板:

template<class InIt, class T, class Pred>

T accumulate(InIt first, InIt last, T val, Pred pr);

- pr -- **函数对象**

  对[first, last)中的每个迭代器 I,

  执行 **val = pr(val, * I)**, 返回最终的val

- pr也可以是个函数名/函数指针

Dev C++ 中的 Accumulate 源代码1:

```
//没有函数对象的版本,仅实现累加
   template<typename _InputIterator,  typename _Tp>
   _Tp  accumulate(_InputIterator __first,  _InputIterator __last,
                        _ Tp __init)
   {

        for ( ; __first != __last; ++__first)
             __init = __init + *__first;
        return __init;
   }

// typename 等效于class
```

Dev C++ 中的 Accumulate 源代码2:

```
//有函数对象的版本,根据函数对象定义的方式实现累加
template<typename _InputIterator,  typename _Tp,  typename
_BinaryOperation>
_Tp accumulate( _InputIterator __first,  _InputIterator __last,
                    _Tp __init, _BinaryOperation __binary_op)
{
    for ( ; __first != __last; ++__first)
        __init = __binary_op(__init,  *__first);
    return __init;
}
```

- 调用accumulate时,和__binary_op对应的实参可以是 函数/函数对象

```
#include <iostream>
#include <vector>
#include <numeric> //accumulate在此文件定义
#include <iterator>
using namespace std;
int sumSquares( int total,  int value)
{
    return total + value * value;
}
```

```
template<class T>
class SumSquaresClass{
    public:
        const T  operator() ( const T & total,  const T & value) {
                return total + value * value;
        }
};
```

```
//注：  VS中如下代码也可以
template<class T>
class SumSquaresClass{
    public:
        const T & operator() ( const T & total,  const T & value)
            {  return total + value * value;  }
};
```

```
template<class T>
class SumPowers{
    private:
        int power;
    public:
        SumPowers(int p):power(p) { }
        const  T operator( ) ( const T & total,  const T & value) {
        //计算 value的power次方, 加到total上
            T v = value;
            for( int i = 0; i < power - 1; ++ i )
                v = v * value;
            return total + v;
        }
};
```

```
int main()  {
    const int SIZE = 10;
    int a1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> v(a1, a1+SIZE);
    ostream_iterator<int> output(cout, " ");
    cout << "1) ";
    copy(v.begin(), v.end(), output); cout << endl;
    int result = accumulate(v.begin(), v.end(), 0, sumSquares);
    cout << "2) 平方和:    " << result << endl;
    SumSquaresClass<int> s;
    result = accumulate(v.begin(), v.end(), 0, s);  // (1)
    cout << "3) 平方和:    " << result << endl;
    result = accumulate(v.begin(), v.end(), 0, SumPowers<int>(3));
    cout << "4) 立方和:    " << result << endl;
    result = accumulate(v.begin(), v.end(), 0, SumPowers<int>(4));
    cout << "5) 4次方和:    " << result;
    return 0;
}
```

输出:

1) 1 2 3 4 5 6 7 8 9 10

2) 平方和:　385

3) 平方和:　385

4) 立方和:　3025

5) 4次方和:　25333

```
int result = accumulate(v.begin(), v.end(), 0, SumSquares);
```

- 实例化出:

```
int accumulate(vector<int>::iterator first,
               vector<int>::iterator last,
               int init, int ( * op)( int, int))
{
    for ( ; first != last; ++first)
        init = op(init, *first);
    return init;
}
```

```
result=
accumulate(v.begin(),  v.end(),  0,
    SumSquaresClass<int>());
//(1)效果一样
```

STL 的**<functional>** 里还有以下**函数对象类模板**:

- **equal_to**
- **greater**
- **less**
- **…**

这些模板可以用来生成**函数对象**

函数对象的参数

根据函数对象参数个数, STL算法用到的主要有三类基类:

- 没有参数的函数对象, 相当于 "f()", 例如:

```
vector<int> V(10);
generate(V.begin(), V.end(), rand);
```

- 一个参数的函数对象, 相当于 "f(x)"

  STL中用unary_function定义了一元函数基类

```
template <class _Arg,  class _Result>
struct unary_function {
    typedef  _Arg argument_type;
    typedef  _Result result_type;
};
```

函数对象的参数

- 两个参数的函数对象, 二元函数, 相当于 "f(x, y)", binary_function定义了二元函数基类:

```
template <class _Arg1,  class _Arg2,  class _Result>
struct binary_function {
    typedef_Arg1 first_argument_type;
    typedef_Arg2 second_argument_type;
    typedef_Result result_type;
};
```

greater 函数对象类模板

例： **greater函数**对象类模板

```
template<class T>
struct greater : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const {
        return x > y;
    }
};
```

greater 的应用

list 有**两个sort函数**

- 前面例子中看到的是**不带参数的**sort函数，
  将list中的元素按 < 规定的比较方法**升序**排列
- list还有另一个sort函数：
  void sort (greater<T> pr);
- 可以用来进行**降序**排序

```
#include <list>
#include <iostream>
#include <iterator>
using namespace std;
int main(){
    const int SIZE = 5;
    int a[SIZE] = {5, 1, 4, 2, 3};
    list<int> lst(a, a+SIZE);
    lst.sort(greater<int>());     // greater<int>()是个对象
                                  //本句进行降序排序
    ostream_iterator<int> output(cout, ", ");
    copy( lst.begin(), lst.end(), output);
    cout << endl;
    return 0;
}
```

输出: 5, 4, 3, 2, 1,

Thanks!