



# 程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>



# 标准模板库STL

## 概述

# 泛型程序设计

C++ 语言的核心优势之一就是便于软件的重用

C++中有两个方面体现重用：

1. 面向对象的思想：继承和多态，标准类库

2. 泛型程序设计(generic programming) 的思想： 模板机制，以及标准模板库 STL

# 泛型程序设计

简单地说就是使用模板的程序设计法。

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，以后则不论数据结构里放的是什么对象，算法针对什么样的对象，则都不必重新实现数据结构，重新编写算法。

**标准模板库**（Standard Template Library）就是一些常用**数据结构和算法的模板的集合**。

有了STL，不必再写大量的标准数据结构和算法，并且可获得非常高的性能。

# STL中的基本的概念

**容器：**可容纳各种数据类型的通用数据结构, 是类模板

**迭代器：**可用于依次存取容器中元素, 类似于指针

**算法：**用来操作容器中的元素的函数模板

- `sort()` 来对一个vector中的数据进行排序
- `find()` 来搜索一个list中的对象

算法本身与他们操作的数据的类型无关, 因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用。

# STL中的基本的概念

```
int array[100];
```

该数组就是容器，而 `int *` 类型的指针变量就可以作为迭代器，`sort` 算法可以作用于该容器上，对其进行排序：

```
sort(array, array+70); //将前70个元素排序
```



迭代器

迭代器

# 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，都是类模版，分为三种：

## 1) 顺序容器

vector, deque, list

## 2) 关联容器

set, multiset, map, multimap

## 3) 容器适配器

stack, queue, priority\_queue

# 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，比如排序，查找，要求对容器中的元素进行比较，有的容器本身就是排序的，所以，放入容器的对象所属的类，往往还应该重载 `==` 和 `<` 运算符。

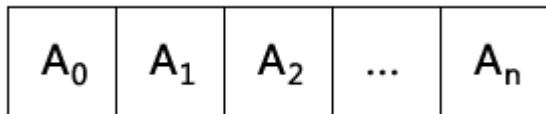
# 顺序容器简介

容器并非排序的，元素的插入位置同元素的值无关。

有vector, deque, list 三种

● **vector** 头文件 `<vector>`

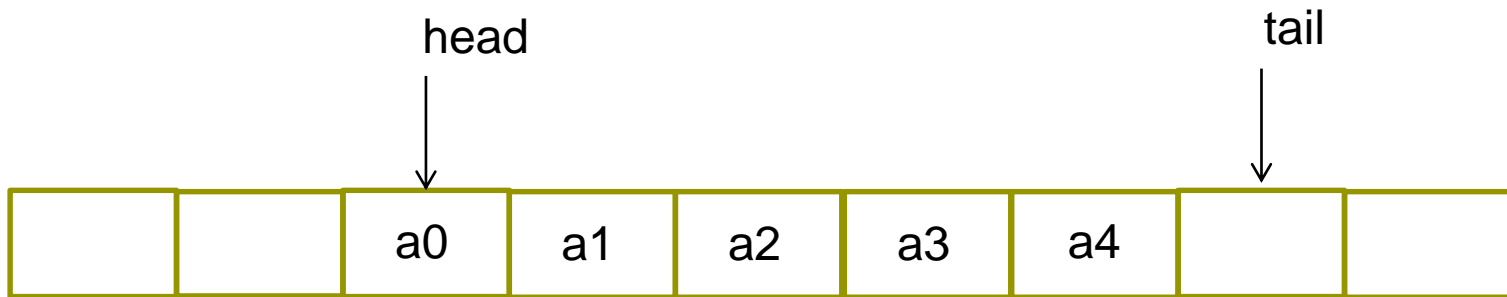
动态数组。元素在内存连续存放。随机存取任何元素都能在**常数时间**完成。在尾端增删元素具有较佳的性能(大部分情况下是常数时间)。



# 顺序容器简介

- deque 头文件 <deque>

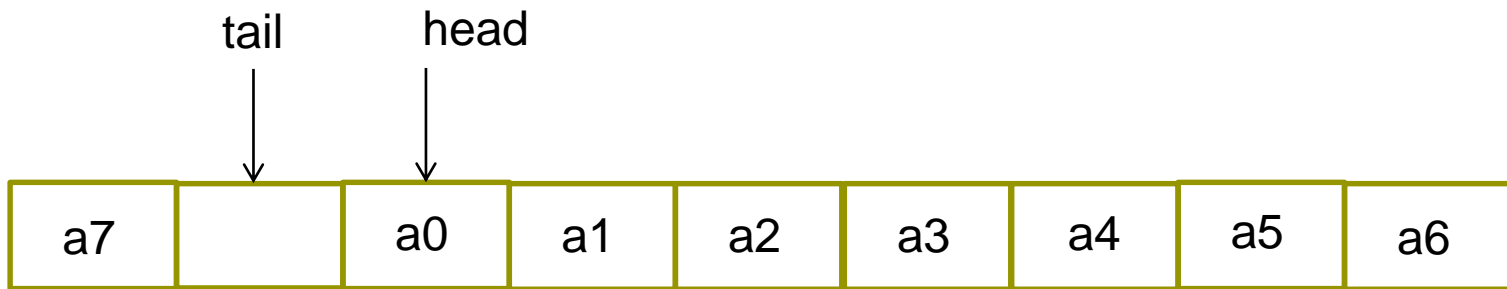
双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成(但次于vector)。在两端增删元素具有较佳的性能(大部分情况下是常数时间)。



# 顺序容器简介

- deque 头文件 <deque>

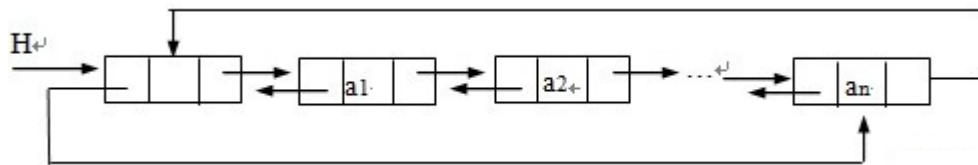
双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成(但次于vector)。在两端增删元素具有较佳的性能(大部分情况下是常数时间)。



# 顺序容器简介

## ● list 头文件 <list>

双向链表。元素在内存不连续存放。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。



# 关联容器简介

- 元素是排序的
- 插入任何元素，都按相应的排序规则来确定其位置
- 在查找时具有非常好的性能
- 通常以平衡二叉树方式实现，插入和检索的时间都是  $O(\log(N))$

- set/multiset 头文件 <set>

set 即集合。set中不允许相同元素，multiset中允许存在相同的元素。

- map/multimap 头文件 <map>

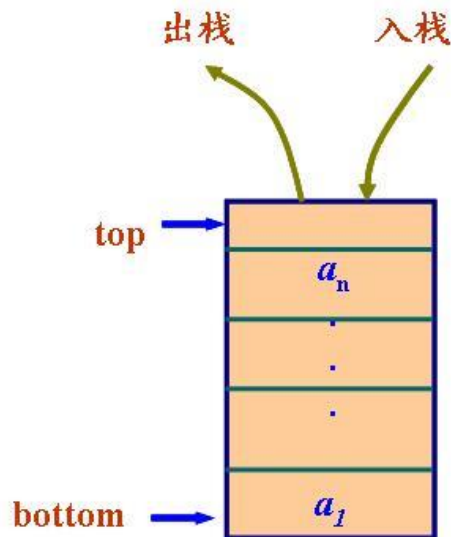
map与set的不同在于map中存放的元素有且仅有两个成员变量，一个名为first，另一个名为second，map根据first值对元素进行从小到大排序，并可快速地根据first来检索元素。

map同multimap的不同在于是否允许相同first值的元素。

# 容器适配器简介

- **stack** : 头文件 `<stack>`

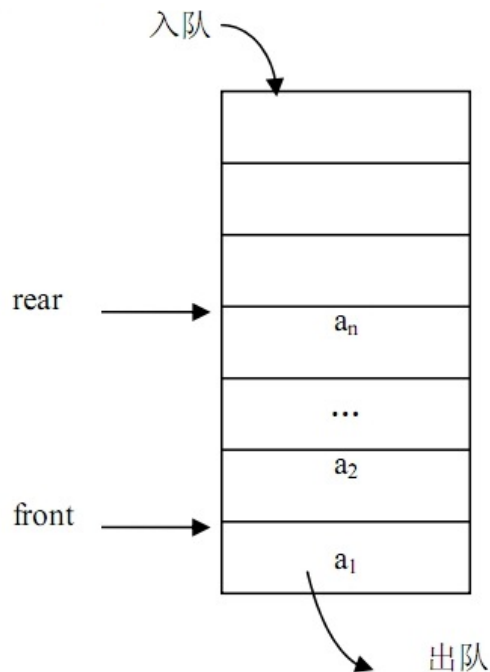
栈。是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项（栈顶的项）。**后进先出**。



# 容器适配器简介

- **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。**先进先出**。



# 容器适配器简介

- `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

## 顺序容器和关联容器中都有的成员函数

- 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

# 顺序容器的常用成员函数

**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

**erase** : 删除迭代器指向的元素(可能会使该迭代器失效)，或删除一个区间，返回被删除元素后面的那个元素的迭代器

# 迭代器

- 用于指向顺序容器和关联容器中的元素
- 迭代器用法和指针类似
- 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

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

迭代器上可以执行 ++ 操作，以使其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，此时再使用它，就会出错，类似于使用NULL或未初始化的指针一样。

# 迭代器示例

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

```
int main() {
```

```
    vector<int> v; //一个存放int元素的数组，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;
```

输出结果:

1,2,3,4,

4,3,2,1,

100,100,100,100,

```
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,

# 双向迭代器

若p和p1都是双向迭代器，则可对p、p1可进行以下操作：

`++p, p++`      使p指向容器中下一个元素

`--p, p--`      使p指向容器中上一个元素

`* p`      取p指向的元素

`p = p1`      赋值

`p == p1 , p != p1`      判断是否相等、不等

# 随机访问迭代器

若 $p$ 和 $p1$ 都是随机访问迭代器，则可对 $p$ 、 $p1$ 可进行以下操作：

➤ 双向迭代器的所有操作

➤  $p += i$  将 $p$ 向后移动 $i$ 个元素

➤  $p -= i$  将 $p$ 向向前移动 $i$ 个元素

➤  $p + i$  值为：指向  $p$  后面的第 $i$ 个元素的迭代器

➤  $p - i$  值为：指向  $p$  前面的第 $i$ 个元素的迭代器

➤  $p[i]$  值为： $p$ 后面的第 $i$ 个元素的引用

➤  $p < p1$ ,  $p \leq p1$ ,  $p > p1$ ,  $p \geq p1$

# In-Video Quiz

1. map、set、list、vector、deque这五种容器中，有几种是排序的？

A) 1 B) 2 C) 3 D) 4

2. 下面5种操作，有几种时间复杂度一定是常数的？

(1)在vector尾部增删元素

(2)在关联容器中查找元素

(3)在list中的某一位置增删元素

(4)随机访问vector中的元素

(5)在multimap中插入元素

A) 1 B) 2 C) 3 D) 4



# 程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>



# 标准模板库STL

## 概述

## 容器

vector

deque

list

set/multiset

map/multimap

stack

queue

priority\_queue

## 容器上的迭代器类别

随机访问

随机访问

双向

双向

双向

不支持迭代器

不支持迭代器

不支持迭代器

## 容器

vector

deque

list

set/multiset

map/multimap

stack

queue

priority\_queue

## 容器上的迭代器类别

随机访问

随机访问

双向

双向

双向

不支持迭代器

不支持迭代器

不支持迭代器

有的算法，例如sort，binary\_search需要通过随机访问迭代器来访问容器中的元素，那么list以及关联容器就不支持该算法！



vector的迭代器是随机迭代器，

遍历 vector 可以有以下几种做法 (deque亦然)：

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

```
int i;
```

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

```
    cout << v[i]; //根据下标随机访问
```

```
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;
```

```
}
```

list 的迭代器是双向迭代器,

正确的遍历list的方法:

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

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

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

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

错误的做法:

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

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

//双向迭代器不支持 <, list没有 [] 成员函数

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

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



ERROR

# 算法简介

- 算法就是一个个函数模板，大多数在`<algorithm>` 中定义
- STL中提供能在各种容器中通用的算法，比如查找，排序等
- 算法通过迭代器来操纵容器中的元素。许多算法可以对容器中的一个局部区间进行操作，因此需要两个参数，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。比如，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，也可以处理普通数组

# 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 `[first, last)`。区间的起点是位于查找范围之中的，而终点不是。**`find`在 `[first, last)` 查找等于 `val` 的元素**
- 用 **`==`** 运算符判断相等
- 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**等于 `last`**

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main() { //find算法示例
    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; //输出3
```

输出:  
3  
not found  
3  
20

```
p = find(v.begin(),v.end(),9);
```

```
if( p == v.end())
```

```
    cout << "not found " << endl;
```

```
p = find(v.begin()+1,v.end()-2,1); //整个容器：[1,2,3,4], 查找区间：[2,3)
```

```
if( p != v.end())
```

```
    cout << * p << endl;
```

```
int * pp = find( array,array+4,20); //数组名是迭代器
```

```
cout << * pp << endl;
```

```
}
```

输出：

3

not found

3

20

# STL中“大”“小”的概念

- 关联容器内部的元素是从小到大排序的
- 有些算法要求其操作的区间是从小到大排序的，称为“有序区间算法”  
例：binary\_search
- 有些算法会对区间进行从小到大排序，称为“排序算法”  
例：sort
- 还有一些其他算法会用到“大”，“小”的概念  
使用STL时，在缺省的情况下，以下三个说法等价：
  - 1) x比y小
  - 2) 表达式“ $x < y$ ”为真
  - 3) y比x大

# STL中“相等”的概念

- 有时，“x和y相等”等价于“**x==y为真**”

例：在未排序的区间上进行的算法，如顺序查找find

.....

- 有时“x和y相等”等价于“**x小于y和y小于x同时为假**”

例：

有序区间算法，如binary\_search

关联容器自身的成员函数find

.....

# 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;
}
```

# 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));
    return 0;
}
```

输出结果:

3<9?

2<9?

1<9?

9<1?

1

# In-Video Quiz

1. 假设p和p1是set上的迭代器，以下5个表达式有几个是非法的？

(1) ++p

(2) --p

(3) p+1

(4) p<p1

(5) p[2]

A) 1 B) 2 C) 3 D) 4

2. binary\_search在查找过程中，比较元素和被查找的值是否相等时，用哪个运算符进行比较？

A) = B) == C) < D) < 和 ==，一个都不能少

# vector

郭 炜 刘家瑛





# vector

- ▀ 可变长的动态数组
- ▀ 必须包含头文件 `#include <vector>`
- ▀ 支持 随机访问迭代器
  - 根据下标随机访问某个元素时间为常数
  - 在尾部添加速度很快
  - 在中间插入慢
- ▀ 所有STL算法 都能对vector操作



# vector的成员函数

## 构造函数初始化

成员函数	作 用
<code>vector();</code>	无参构造函数, 将容器初始化成空的
<code>vector(int n);</code>	将容器初始化成有n个元素
<code>vector(int n, const T &amp; val);</code>	假定元素类型是T, 将容器初始化成有n个元素, 每个元素的值都是val
<code>vector(iterator first, iterator last);</code>	将容器初始化为与别的容器上区间[first, last)一致的内容



# vector的成员函数

## 其他常用函数

成员函数	作 用
<code>void pop_back();</code>	删除容器末尾的元素
<code>void push_back(const T &amp; val);</code>	将val添加到容器末尾
<code>int size();</code>	返回容器中元素的个数
<code>T &amp; front();</code>	返回容器中第一个元素的引用
<code>T &amp; back();</code>	返回容器中最后一个元素的引用



## 例1:

```
#include <iostream>
#include <vector>
using namespace std;
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.at(i) << "," ;
}
```

输出:

5

0,1,2,3,100,

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




## 二维动态数组

```
vector< vector<int> > v(3);
```

//v有3个元素,

//每个元素都是vector<int> 容器



## 例2:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<vector<int >> v(3);
    for(int i=0; i<v.size(); ++i)
        for(int j=0; j<4; ++j)
            v[i].push_back(j);
    for(int i=0; i<v.size(); ++i){
        for(int j=0; j<v[i].size(); ++j)
            cout<<v[i][j]<<" ";
        cout<<endl;
    }
    return 0;
}
```

输出:

0 1 2 3

0 1 2 3

0 1 2 3



# In-Video Quiz

vector没有以下哪个成员函数：

- ▲ A) push\_back
- ▲ B) front
- ▲ C) pop\_back
- ▲ D) push\_front

# list 和 deque

郭 炜 刘家瑛



北京大學

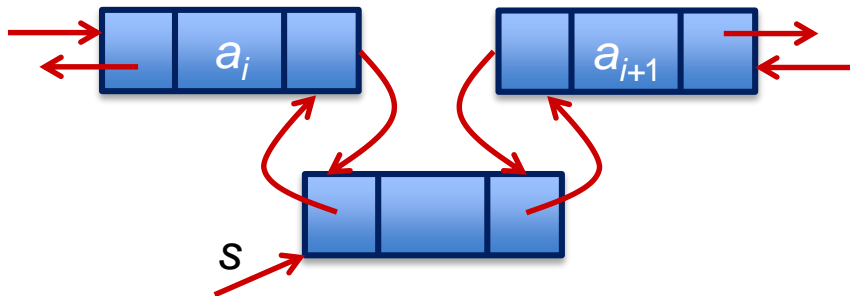


# list 容器

- 双向链表 `#include <list>`



- 在任何位置插入/删除都是**常数时间**



- 不支持根据下标随机存取元素
- 具有所有顺序容器都有的成员函数



# list 容器

## 还支持8个成员函数:

成员函数	作 用
push_front	在链表 <b>最前面插入</b>
pop_front	删除链表 <b>最前面</b> 的元素
sort	排序 ( <b>list 不支持 STL 的算法 sort</b> )
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, 并清空被合并的链表
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

→ 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
```

- 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;
```

```
        i = lst.begin();
```

```
        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;  
}
```

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,



# deque 容器

- ▀ 双向队列
- ▀ 必须包含头文件 `#include <deque>`
- ▀ 所有适用于vector的操作 → 都适用于deque
- ▀ deque还有 `push_front` (将元素插入到容器的头部) 和 `pop_front` (删除头部的元素) 操作



# In-Video Quiz

deque没有以下哪个成员函数

- ▲ A) sort
- ▲ B) push\_front
- ▲ C) pop\_front
- ▲ D) back



# 程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

刘家瑛 微博 <http://weibo.com/pkuliujiaying>



# 标准模板库STL

## 函数对象

# 函数对象

- 若一个类重载了运算符 “()”,  
则该类的对象就成为函数对象

```
class CMyAverage { //函数对象类
```

```
public:
```

```
    double operator() ( int a1, int a2, int a3 ) {
```

```
        return (double)(a1 + a2+a3) / 3;
```

```
    }
```

```
};
```

```
CMyAverage average; //函数对象
```

```
cout << average(3,2,3); // average.operator() (3, 2, 3)
```

输出 2.66667

# 函数对象的应用

Dev C++ 中的 Accumulate 源代码1:

```
template<typename _InputIterator, typename _Tp>
_Tp accumulate(_InputIterator __first, _InputIterator __last,
               _Tp __init)
{
    for ( ; __first != __last; ++__first)
        __init = __init + *__first;
    return __init;
}
```

// typename 等价于class

# 函数对象的应用

Dev C++ 中的 Accumulate 源代码2:

```
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或函数对象

# 函数对象的应用示例

```
#include <iostream>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
#include <numeric>
```

```
#include <functional>
```

```
using namespace std;
```

```
int sumSquares( int total, int value)
```

```
{    return total + value * value; }
```

```
template <class T>
```

```
void PrintInterval(T first, T last)
```

```
{ //输出区间[first, last)中的元素
```

```
    for( ; first != last; ++ first)
```

```
        cout << * first << " ";
```

```
    cout << endl;
```

```
}
```

```
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);
    cout << "1) "; PrintInterval(v.begin(),v.end());
    int result = accumulate(v.begin(),v.end(),0,SumSquares);
    cout << "2) 平方和: " << result << endl;
    result =
        accumulate(v.begin(),v.end(),0,SumPowers<int>(3));
    cout << "3) 立方和: " << result << endl;
    result =
        accumulate(v.begin(),v.end(),0,SumPowers<int>(4));
    cout << "4) 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) 立方和: 3025
4) 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;
}
```

```
accumulate(v.begin(),v.end(),0,SumPowers<int>(3));
```

实例化出：

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

# STL中的函数对象类模板

以下模板可以用来生成函数对象。

equal\_to

greater

less

.....

头文件： <functional>

# 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成员函数

➤ void sort();

将list中的元素按 “<” 规定的比较方法升序排列。

➤ template <class Compare>

void sort (Compare op);

将list中的元素按 op 规定的比较方法升序排列。即要比较x,y大小时，看 **op(x,y)** 的返回值，为true则认为 x小于y

# greater 的应用

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

class MyLess {
public:
    bool operator()( const int & c1, const int & c2 )
    {
        return (c1 % 10) < (c2 % 10);
    }
};

template <class T>
void Print(T first,T last) {
    for( ; first != last ; ++ first ) cout << * first << ",";
}
```

```
int main()
{
    const int SIZE = 5;
    int a[SIZE] = {5,21,14,2,3};
    list<int> lst(a,a+SIZE);
    lst.sort(MyLess());
    Print( lst.begin(),lst.end());
    cout << endl;
    lst.sort(greater<int>()); //greater<int>() 是个对象
    Print( lst.begin(),lst.end());
    cout << endl;
    return 0;
}
```

输出：

21,2,3,14,5,  
21,14,5,3,2,

## 在STL中使用自定义的“大”，“小”关系

关联容器和STL中许多算法，都是可以用函数或函数对象自定义比较器的。在自定义了比较器`op`的情况下，以下三种说法是等价的：

- 1) `x`小于`y`
- 2) `op(x,y)`返回值为`true`
- 3) `y`大于`x`

## 例题:写出MyMax模板

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

class MyLess {
public:
    bool operator() (int a1,int a2) {
        if( ( a1 % 10 ) < (a2%10) )
            return true;
        else
            return false;
    }
};

bool MyCompare(int a1,int a2)
{
    if( ( a1 % 10 ) < (a2%10) )
        return false;
    else
        return true;
}
```

```
int main()
{
    int a[] = {35,7,13,19,12};
    cout << * MyMax(a,a+5,MyLess())
            << endl;
    cout << * MyMax(a,a+5,MyCompare)
            << endl;
    return 0;
}
```

输出:  
19  
12

## 例题:写出MyMax模板

```
template <class T, class Pred>
T MyMax( T first, T last, Pred myless)
{
    T tmpMax = first;
    for(; first != last; ++ first)
        if( myless( * tmpMax,* first))
            tmpMax = first;
    return tmpMax;
};
```

## In-Video Quiz

如果A是一个类，lst是list<A>类的对象，则若要使lst.sort(greater<A>()) 能够工作，必须重载哪个运算符？

- A) >
- B) <
- C) ==
- D) 亲，根本不需要重载任何运算符的哟