



程序设计艺术与方法学

第一讲 STL简介

一个问题：输入任意个整数，排序然后输出。





Ce

川大智胜杯全国大学

2005

银



acm



Wenxin Li

Dr. Wenxin Li
ACM/ICPC As



四川大学

ACM/ICPC2006亚洲区预选赛（北京）
暨Google杯全国大学生程序设计邀请赛

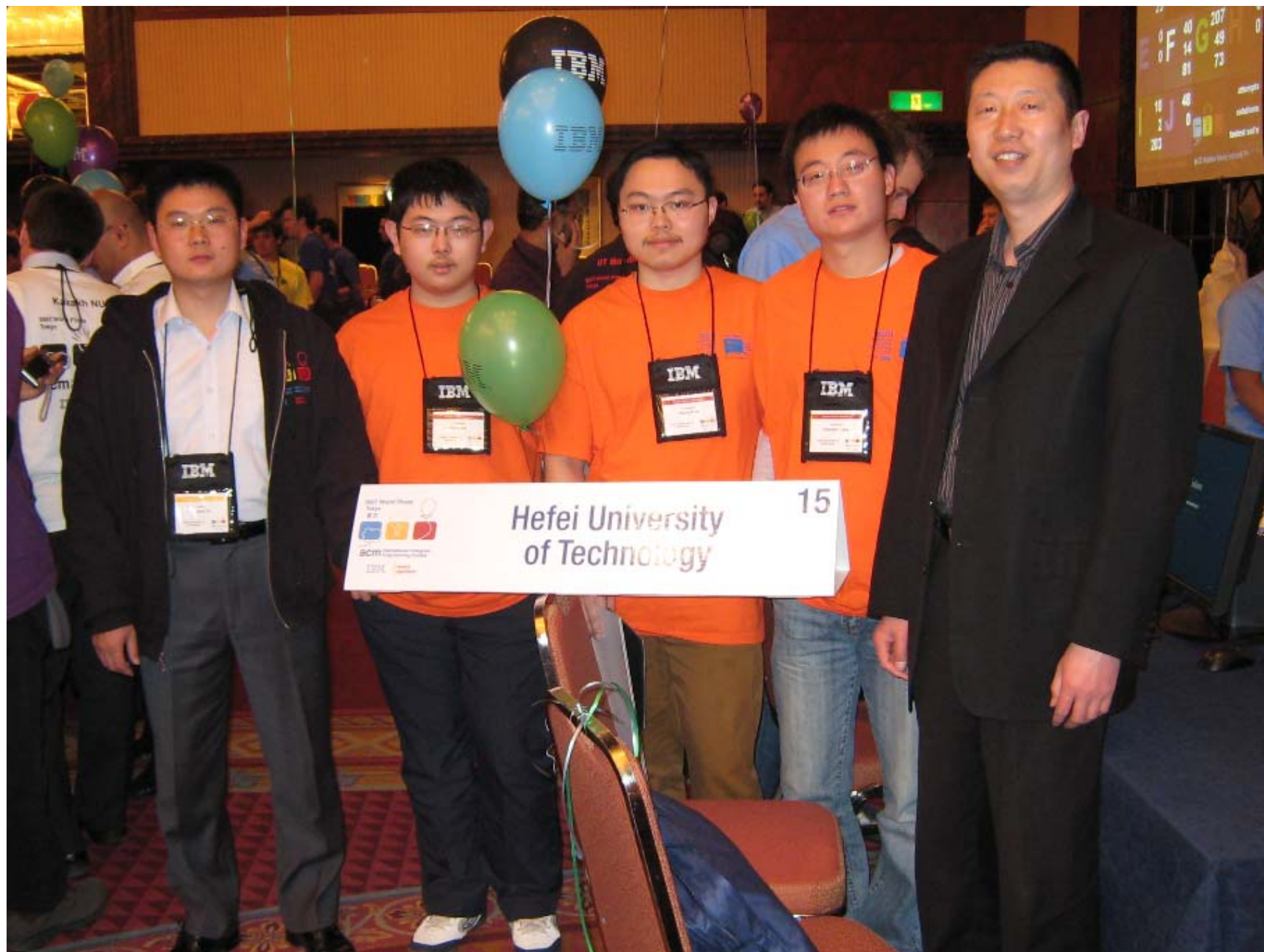
金奖



清华大学

Google
谷歌







第一章 STL简介



1.1 引言

1.2 STL的组成结构

1.3 STL的应用



1.1 引言



- ⊕ **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：
 - 面向对象的思想：继承和多态，标准类库。
 - 泛型程序设计(**generic programming**) 的思想：模板机制，以及标准模板库**STL**。

标准模板库 (Standard Template Library) 是ANSI/ISO C++语言的库的一个主要组成部分。它包括了通用数据结构和基于这些结构的算法，向外提供统一标准的公共接口，使得使用STL方便、快捷地建立应用程序。



泛型程序设计

- 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。
- 有了**STL**，不必再从头写太多的标准数据结构和算法，并且可获得非常高的性能。



1.1 引言



假如设计一个求两参数最大值的函数，在实践中我们可能需要定义四个函数：

```
int max (int a, int b) { return (a>b) ? a : b; }
```

```
long max (long a , long b ) { return ( a > b ) ? A : b ;}
```

```
double max (double a , double b ) { return ( a > b ) ? A : b ; }
```

```
char max (char a , char b ) { return ( a > b ) ? a : b ;}
```

- ⊕ 这些函数几乎相同，唯一的区别就是形参类型不同；
- ⊕ 需要事先知道有哪些类型会使用这些函数，对于未知类型这些函数不起作用。



模板的概念

1. 所谓模板是一种使用无类型参数来产生一系列**函数或类**的机制。
2. 若一个程序的功能是对某种特定的数据类型进行处理，则可以将所处理的数据类型说明为参数，以便在其他数据类型的情况下使用，这就是**模板的由来**。
3. 模板是以一种完全通用的方法来设计函数或类而**不必预先说明**将被使用的每个对象的类型。
4. 通过模板可以产生类或函数的集合，使它们操作不同的数据类型，从而**避免**需要为每一种数据类型产生一个单独的类或函数。



1.1 引言



求最大值模板函数实现

1.求两个数最大值，使用模板

```
template < class T >  
T max(T a , T b){  
    return ( a > b ) ? a : b;  
}
```

2.template < 模板形参表>

```
<返回值类型> <函数名> (模板函数形参表)  
{  
    //函数定义体  
}
```



模板工作方式

- ⊕ 函数模板只是说明，不能直接执行，需要实例化为模板函数后才能执行；
- ⊕ 在说明了一个函数模板后，当编译系统发现有一个对应的函数调用时，将根据实参中的类型来确认是否匹配函数模板中对应的形参，然后生成一个重载函数。该重载函数的定义体与函数模板的函数定义体相同，它称之为**模板函数**。



1.1 引言

编写一个对具有**n**个元素的数组**a[]**求最小值的程序，要求将求最小值的函数设计成函数模板。 (例1)

```
#include <iostream>
template <class T>
T min(T a[],int n)
{
    int i;
    T minv=a[0];
    for( i = 1;i < n ; i++) {
        if(minv>a[i])
            minv=a[i];
    }
    return minv;
}
```

```
int main()
{ int a[]={1,3,0,2,7,6,4,5,2};
  double b[]={1.2,-3.4,6.8,9,8};
  cout<<" a数组的最小值为: "
  <<min(a,9)<< endl;
  cout<<" b数组的最小值为: "
  <<min(b,4)<<endl; }
```

此程序的运行结果为:

a数组的最小值为: 0

b数组的最小值为: -3.4



1.1 引言



模板优缺点

- ⊕ 函数模板方法克服了**C**语言解决上述问题时用大量不同函数名表示相似功能的坏习惯；
- ⊕ 克服了宏定义不能进行参数类型检查的弊端；
- ⊕ 克服了**C++**函数重载用相同函数名字重写几个函数的繁琐。
- ⊕ 缺点，调试比较困难。
 - 一般先写一个特殊版本的函数；
 - 运行正确后，改成模板函数。



第一讲 STL简介



1.1 引言

1.2 STL的组成结构

1.3 STL的应用



什么是STL?



- ⊕ **STL全名标准模版库(Standard Template Library)**, 是一群以**template**为根基的**C++**程序库
- ⊕ 目的在提供一些基本的容器类别(container class)与高效的算法(algorithm)
- ⊕ 一般来说程序是由算法加上数据结构, 互相配合、一起工作, 完成程序的功能。但如此一来便将数据结构与算法紧密绑定, 不能分离, 缺乏弹性
- ⊕ **Stepanov**先生便针对此问题, 采通用型设计的思维, 使得程序员可以:



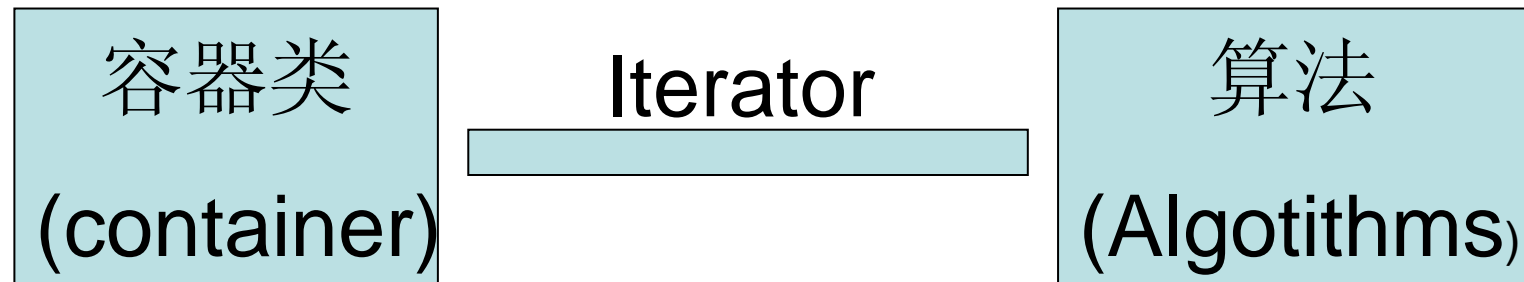
1. 以有效率的算法解决问题，此算法可处理各种数据结构，即算法与数据结构互为独立
2. 各种数据结构使用一致的接口(**interface**)，让各种算法可以透过此接口处理各种数据结构



1.2 STL的组成结构



算法 + 数据结构 = 程序

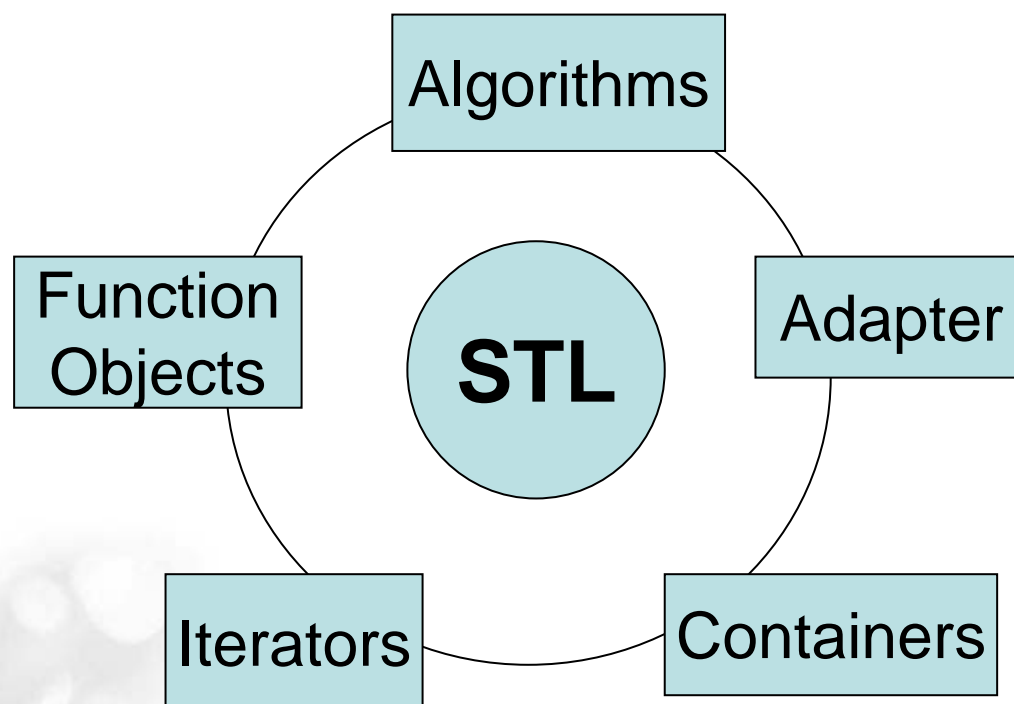




1.2 STL的组成结构



算法 + 数据结构 = 程序





1.2 STL的组成结构



- ⊕ 容器 (**container**)—用来容纳各种数据类型的数据结构;
- ⊕ 迭代器(**iterator**)—好比传统C语言的指针, 可藉之依次存取容器中元素的对象;
- ⊕ 算法(**algorithm**)—算法通过迭代器来操作容器中元素的函数模版;
- ⊕ 适配器(**adaptor**)—利用基础容器对象, 加以包装, 改变其接口, 以适应另一种需求;
- ⊕ 函数对象(**function object**)—为STL中较低阶的对象, 用来代替传统的函数指针(**function pointer**)。



1.2 STL的组成结构



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

vector: 后部插入/删除，直接访问

deque: 前/后部插入/删除，直接访问

list: 双向链表，任意位置插入/删除

2) 关联容器

set: 快速查找，无重复元素

multiset: 快速查找，可有重复元素

map: 一对一映射，无重复元素，基于关键字查找

multimap: 一对一映射，可有重复元素，基于关键字查找

前2者合称为第一类容器

3) 容器适配器

stack: LIFO

queue: FIFO

priority_queue: 优先级高的元素先出



1.2 STL的组成结构



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(但性能次于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 STL的组成结构



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>

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

2) map/multimap: 头文件 <map>

➤ **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多个元素有相同的key值。

上述4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 $O(\log N)$



1.2 STL的组成结构



容器适配器简介

1) stack: 头文件 <stack>

- 栈。是个有限序列，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。



容器的共有成员函数

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大小的运算符：**=, <, <=, >, >=, ==, !=**
- **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
- **begin、end、insert、erase**等



1.2 STL的组成结构



vector的成员函数 (例2)

函数名	返回值	参数	功能
at	元素的引用	要取元素的位置(int pos)	如果该位置元素存在则返回它的引用，否则进行一场处理
back	元素的引用	无	返回容器最后一个元素的引用，如果容器为空则出错
begin	迭代器	无	取得元素的首迭代器
end	迭代器	无	取得元素的尾迭代器
clear	无	无	将容器清空
empty	bool值	无	如果容器为空则为真，否则为假
erase	迭代器	要删除元素的迭代器 要删除容器段的首尾迭代器	第一个版本删除指定的元素 第二个版本删除指定的容器段，不存在则出错
front	元素的引用	无	返回首元素的引用，为空则出错
insert	无	iterator it, const T& x 2、iterator it, const_iterator first, const_iterator last	在it这个位置插入元素T 从it位置开始，依次插入first到last所指的元素
pop_back	无	无	删除最后一个元素
push_back	无	const T& x	从最后插入一个元素
resize	无	size_type n	将容器大小重设置为n
size	int	无	返回容器中元素的个数



1.2 STL的组成结构



再看一个**map**的例子(例3)



⊕ 示例

- 给定一个存储整数的**vector**及一个整数值，在**vector**中查找这个值，并返回指针指向该值，**null**表示不存在该值

⊕ 一个解：

```
const int* find( const vector<int>& vec, int value)
{
    for (int i=0; i<vec.size(); ++i)
        if ( vec[i] == value )
            return &vec[i];
    return null;
}
```



指针的抽象(2)



- ✦ 变化（增加对**float**的功能）：**template**形式

```
Template <typename elemType>
const elemType * find( const vector< elemType >& vec, elemType value)
{
    for (int i=0; i<vec.size(); ++i)
        if ( vec[i] == value )
            return &vec[i];
    return null;
}
```

- ✦ 进一步要求：可以同时处理**array**
 - 其一：通过函数的**overload**实现
 - 另一个：只写一份，同时处理**vector**和**array**
(如何实现?)



⊕ 对策：问题分解

- 一、**array**传入**find()**不实际指明**array**
- 二、**vector**传入**find()**不实际指明**vector**

⊕ 首先处理**array**的问题

- **array**如何传入函数及如何返回？

```
int min( int array[24] ) { ... }
```

实际相当于

```
int min( int* array) { ... }
```

- **24**怎么办？



指针的抽象(4)



- ✚ 解法一，增加一个参数表示**array**的容量

```
template <typename elemType>
const elemType* find( const elemType* array, int size,
                     const elemType& value );
```

- ✚ 解法二，传入另一个地址，作为**array**的终点

```
template <typename elemType>
const elemType* find( const elemType* array,
                     const elemType* sentinel,
                     const elemType& value );
```

- ✚ 🤖 **array**从参数表中消失了，第一个问题解决



⊕ 实现

```
template <typename elemType>
const elemType* find( const elemType* first,
                     const elemType* last,
                     const elemType& value );
{
    if ( !first || !last ) return null;

    for ( ; first != last; ++first )
        if ( *first == value )
            return first;

    return null;
}
```



Φ 使用

```
int ia[8] = { 3, 5, 67, 2, 46, 7, 53, 67 };  
double da[8] = { 25.1, 1.2, 58.0, 25.3, 13.5, 46.0, 25, 58.8 };  
string sa[4] = { "lkei", "iekadkl", "iejg", "ietjpal" };
```

```
const int* pi = find( ia, ia+8, ia[3] );  
const double* pd = find( da, da+8, da[3] );  
const string* ps = find( sa, sa+4, sa[3] );
```

✦ 如何完成第二个子任务？

- **vector**和**array**相同，都是以一块连续内存存储器所有元素，可按**array**相同方式处理
- 用一组表示“开始地址/结束地址”的参数传入**find()**



⊕ 例如

```
vector<string> svec;  
find (&svec[0], &svec[svec.size()], search_value);  
//需要事先确定svec为非空，也许更安全的方法  
if ( !svec.empty() )  
    find (&svec[0], &svec[svec.size()], search_value);
```

更好的方法:

```
template <typename elemType>  
inline elemType* begin(const vector<elemType>& vec )  
{    return vec.empty()? null : &vec[0] ;}  
//同样实现end()  
find (begin(svec), end(svec), search_value);
```

⊕ ☺任务完成

⊕ 可是如果要加上对list的支持呢?

➤ 解决办法是在底层指针指上提供一层抽象化机制



1.2 STL的组成结构

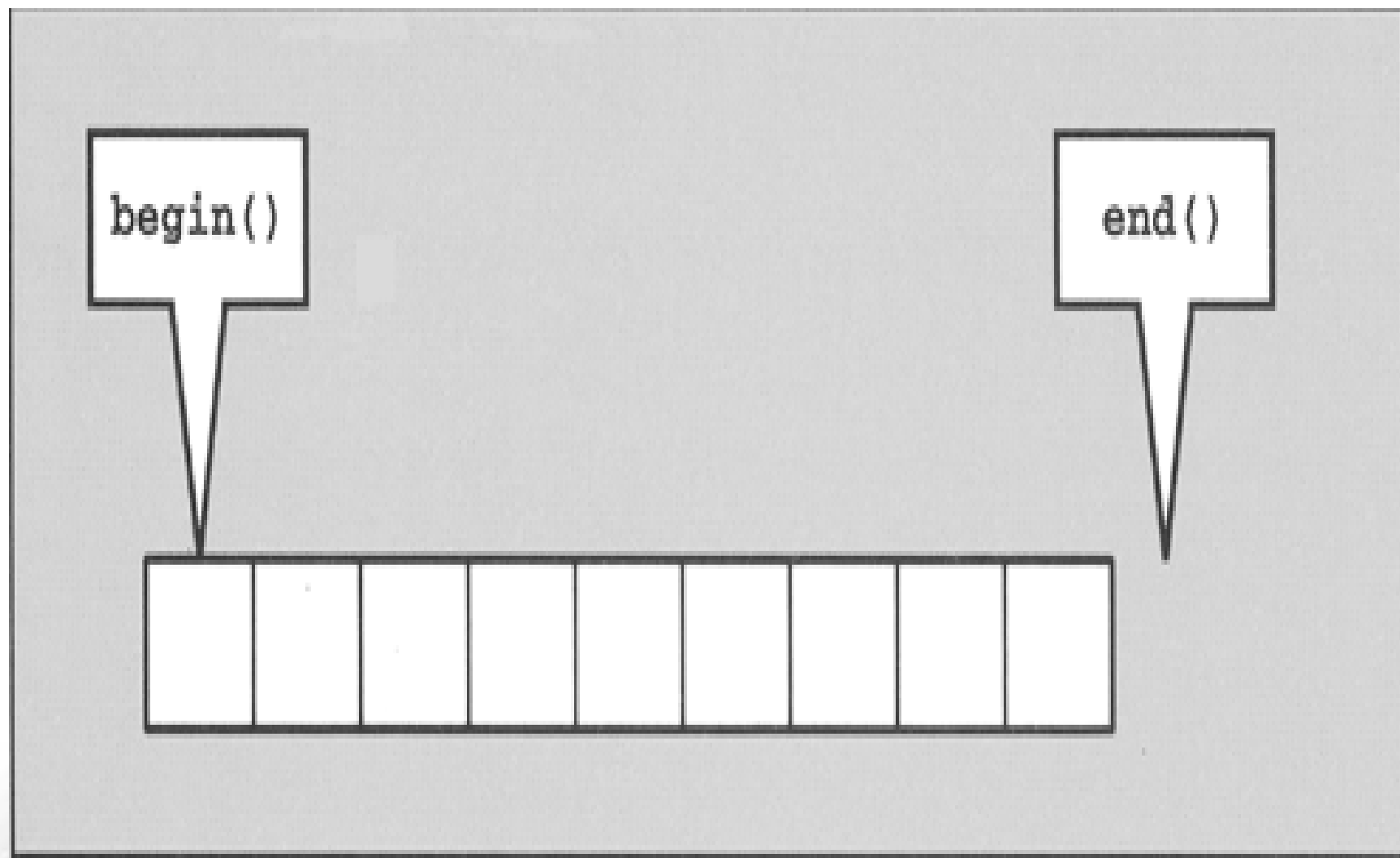


迭代器

- ⊕ 它的很多性质是和**C++/C**中的**指针**是相似，甚至可以说指针就是一种**C++**编译器里内置的迭代器。可以说就是**指针的类实现**；
- ⊕ 软件设计有一个基本原则，所有的问题都可以通过引进一个间接层来简化，这种简化在**STL**中就是用迭代器来完成的；
- ⊕ 迭代器在**STL**中用来将算法和容器联系起来，起着一种**黏和剂**的作用；
- ⊕ 几乎**STL**提供的所有算法都是通过迭代器存取元素序列进行工作的，每一个容器都定义了其本身所专有的迭代器，用以存取容器中的元素。



Iterator的范围表示





1.2 STL的组成结构



迭代器

- ⊕ 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



1.2 STL的组成结构



迭代器

- ✦ 迭代器上可以执行++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, 就好像使用**NULL**或未初始化的指针一样。



1.2 STL的组成结构



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。
 - 例1：只有第一类容器能用迭代器遍历。
 - 例2：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。



1.2 STL的组成结构



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。
 2. 输出: **Output iterators** 提供对数据的只写访问。
 3. 正向: **Forward iterators** 提供读写操作, 并能一次一个地向前推进迭代器。
 4. 双向: **Bidirectional iterators** 提供读写操作, 并能一次一个地向前和向后移动。
 5. 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。



Iterator的分类



Input iterator

Output iterator

Forward iterator

Bidirectional iterator

Random access iterator



1.2 STL的组成结构



迭代器

- ⊕ 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:
 - **`p+= i, p -= i,`**
 - **`p+i:`** 返回指向**p**后面的第**i**个元素的迭代器
 - **`p-i:`** 返回指向**p**前面的第**i**个元素的迭代器
 - **`p[i]:`****p** 后面的第**i**个元素的引用
 - **`p < p1, p <= p1, p > p1, p>= p1`**



1.2 STL的组成结构



迭代器

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



1.2 STL的组成结构



迭代器的例子 (例4)



1.2 STL的组成结构



流迭代器

分为**ostream_iterator**和**istream_iterator**，其中输出流迭代器，可指定写入的目的地和间隔符；输入流可指定读取的来源，并应该和结束迭代器比较。使用迭代器的优点就是可以使用算法，并且很多东西都可抽象为流，确实方便；最经常的当然是**cout**和**cin**。 (例5)



1.2 STL的组成结构



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。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**C语言的数组**。



1.2 STL的组成结构



算法分类

⊕ 变化序列算法

- **copy, remove, fill, replace, random_shuffle, swap, ...**
- 会改变容器

⊕ 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

⊕ 此外还有其他算法，比如**<numeric>**中的算法



1.2 STL的组成结构



排序和查找算法

⊕ **Sort** (例6、7)

- **template<class RanIt>**
- **void sort(RanItfirst, RanItlast);**

⊕ **find** (例8)

- **template<class InIt, class T>**
- **InItfind(InItfirst, InItlast, const T& val);**

⊕ **binary_search**折半查找，要求容器已经有序

- **template<class FwdIt, class T>**
- **boolbinary_search(FwdItfirst, FwdItlast, const T& val);**



第一讲 STL简介



1.1 引言

1.2 STL的组成结构

1.3 STL的应用



1.3 STL的应用



题目：对于输入给定的字符串将其反序输出。
输入是先输入一个整数代表字符串的行数，然后是字符串。

Sample Input:

3

**Frankly, I don't think we'll make much
money out of this scheme.
madam I'm adam**

Sample Output

**hcum ekam ll'ew kniht t'nod I ,ylknarF
.emehcs siht fo tuo yenom
mada m'I madam**