

# STL

(佳木斯大学acm实验室内部资料-----ldc)

**STL.....**  
**.....**

**一，关于**  
**STL.....**  
**.....**

**二，使用**  
**STL.....**  
**.....**

**ACM/ICPC 竞赛之 STL--**  
**pair.....**

**ACM/ICPC 竞赛之 STL--**  
**vector.....**

**ACM/ICPC 竞赛之 STL--**  
**iterator.....**

**ACM/ICPC 竞赛之 STL--**  
**string.....**

**ACM/ICPC 竞赛之 STL--**  
**list.....**

**ACM/ICPC 竞赛之 STL--**  
**stack/queue.....**

**1.stack.....**  
**.....**

**2.queue.....**  
**.....**

### 3.priority\_queue.....

## ACM/ICPC 竞赛之 STL--set.....

### 1.set.....

### 2.multiset.....

## ACM/ICPC 竞赛之 STL--map.....

### 1.map.....

### 2.multimap.....

## ACM/ICPC 竞赛之 STL--algorithm.....

## STL

### 一、关于STL

STL(Standard Template Library, 标准模板库)是C++语言标准中的重要组成部分。STL以模板类和模板函数的形式为程序员提供了各种数据结构和算法的精巧实现,程序员如果能够充分地利用STL,可以在代码空间、执行时间和编码效率上获得极大的好处。

STL大致可以分为三大类: 算法(algorithm)、容器(container)、迭代器(iterator)。

STL容器是一些模板类,提供了多种组织数据的常用方法,例如vector(向量,类似于数组)、list(列表,类似于链表)、deque(双向队列)、set(集合)、map(映象)、stack(栈)、queue(队列)、priority\_queue(优先队列)等,通过模板的参数我们可以指定容器中的元素类型。STL算法是一些模板函数,提供了相当多的有用算法和操作,从简单如for\_each(遍历)到复杂如stable\_sort(稳定排序)。STL迭代器是对C中的指针的一般化,用来将算法和容器联系起来。几乎所有的STL算法都是通过迭代器来存取元素序列进行工作的,而STL中的每一个容器也都定义了其本身所专有的迭代器,用以存取容器中的元素。有趣的是,普通的指针也可以像迭代器一样工作。熟悉了STL后,你会发现,很多功能只需要用短短的几行就可以实现了。通过STL,我们可以构造出优雅而且高效的代码,甚至比你自己手工实现的代码效果还要好。

STL的另外一个特点是,它是以源码方式免费提供的,程序员不仅可以自由地使用这些代码,也可以学习其源码,甚至按照自己的需要去修改它。

## 二. 使用stl

在C++标准中，STL被组织为以下的一组头文件（注意，是没有.h后缀的！）：

algorithm / deque / functional / iterator / list / map

memory / numeric / queue / set / stack / utility / vector

当我们需要使用STL的某个功能时，需要嵌入相应的头文件。但要注意的是，在C++标准中，STL是被定义在std命名空间中的。如下例所示：

---

```
include

main() {

std::stack s;

s.push(0); ...

return 1;

}
```

---

如果希望在程序中直接引用STL，也可以在嵌入头文件后，用using namespace语句将std命名空间导入。如下例所示：

---

```
include

using namespace std;

main() {

stack s;

s.push(0); ...

return 1;

}
```

---

STL是C++语言机制运用的一个典范，通过学习STL可以更深刻地理解C++语言的思想和方法。在本系列的文章中不打算对STL做深入的剖析，而只是想介绍一些STL的基本应用。

有兴趣的同学，建议可以在有了一些STL的使用经验后，认真阅读一下《C++ STL》这本书（电力出版社有该书的中文版）。

## ACM/ICPC 竞赛之 STL--pair

---

### 简介

STL 的头文件中描述了一个看上去非常简单的模板类pair，用来表示一个二元组或元素对，并提供了按照字典序对元素对进行大小比较的比较运算符模板函数。例如，想要定义一个对象表示一个平面坐标点，则可以： pair p1;cin >> p1.first >> p1.second;pair 模板类需要两个参数：首元素的数据类型和尾元素的数据类型。pair 模板类对象有两个成员：first 和second，分别表示首元素和尾元素。

在中已经定义了pair 上的六个比较运算符：<、>、<=、>=、==、!=，其规则是先比较first，first 相等时再比较second，这符合大多数应用的逻辑。当然，也可以通过重载这几个运算符来重新指定自己的比较逻辑。除了直接定义一个pair 对象外，如果需要即时生成一个pair 对象，也可以调用在中定义的一个模板函数：make\_pair。make\_pair 需要两个参数，分别为元素对的首元素和尾元素。

## 一，构造函数

### pair p1;

创建一个空的pair对象，他的两个元素分别是T1类型和T2类型，采用值初始化

### pair p1(v1,v2);

创建一个pair对象，他的两个元素分别是T1类型和T2类型，其中first成员初始化为v1，而second成员初始化为v2

应用：

```
pair<string,string>anon;
pair<string,int>word_count;
pair<string, vector<int> >line;
pair<string,string>author("James","Joy");
```

## 二，相关操作

make\_pair(v1,v2) //在map中应用

以v1，v2的值创建一个新的pair对象，其元素类型分别是v1,v2的类型

p1 < p2

两个pair对象之间小于运算，其定义遵循字典次序：如果p1.first < p2.first 或者！（p2.first < p1.first）&& p1.second < p2.second，则返回true

### p.first

返回名中first的（公有）数据成员

### p.second

返回名中second的（公有）数据成员

应用：

pair类型的使用相当的繁琐，如果定义多个相同的pair类型对象，可以使用typedef简化声明：

```
typedef pair<string,string> Author;
Author proust("March","Proust");
Author Joy("James","Joy");
```

访问两个元素（通过first和second）：

```
pair<int, double> p1; //使用默认构造函数
p1.first = 1;
p1.second = 2.5;
cout << p1.first << ' ' << p1.second << endl;
```

利用make\_pair () 赋值：

```
1    pair<int, double> p1;
2    p1 = make_pair(1, 1.2);
```

变量间赋值：

```
pair<int, double> p1(1, 1.2);
pair<int, double> p2 = p1;
```

来源网址：

<http://www.cnblogs.com/cszlg/archive/2013/03/10/2952807.html>

[http://blog.csdn.net/sinat\\_35121480/article/details/54728594](http://blog.csdn.net/sinat_35121480/article/details/54728594)

## ACM/ICPC 竞赛之 STL--vector

### 简介

在STL的头文件中定义了vector（向量容器模板类），vector容器以连续数组的方式存储元素序列，可以将vector看作是以顺序结构实现的线性表。当我们在程序中需要使用动态数组时，vector将会是理想的选择，vector可以在使用过程中动态地增长存储空间。vector模板类需要两个模板参数，第一个参数是存储元素的数据类型，第二个参数是存储分配器的类型，其中第二个参数是可选的，如果不给出第二个参数，将使用默认的分配器。

### 一，构造函数

**vector () ;**

创建一个空向量。这是一个默认构造函数

**vector (int n, const T &value = T () ) ;**

创建具有n个元素的向量，每个元素具有指定的数值。如果忽略value参数，则元素用类型T的默认值填充。类型T必须有默认的构造函数，类型T的默认值有T () 指定。

**vector (T \* first,T\* last);**

用地址范围 (first, last) 初始化向量。符号first 和last 是指针标志方法。

应用：

```
vector<int> v1(5);           //构造了包含5个默认值的元素的vector

vector<int> v1( 5, 42 );     //构造了一个包含5个值为42的元素的Vector

int a[5] = {1,2,3,4,5};
vector<int> v1(a,a+5);       //从a到a+5复制元素到vector中去
```

## 二，相关操作

### 运算符

语法:

C++ Vectors能够使用标准运算符: ==, !=, <=, >=, <, 和 >.

v[] // 要访问vector中的某特定位置的元素可以使用 [] 操作符.

v1 == v2 // 两个vectors被认为是相等的,如果: 1.它们具有相同的容量;2.所有相同位置的元素相等.

v1 != v2

v1 <= v2 // vectors之间大小的比较是按照词典规则.

v1 >= v2

v1 < v2

v1 > v2

### assign函数

语法:

```
void assign( input_iterator start, input_iterator end );
```

//assign() 函数要么将区间[start, end)的元素赋到当前vector

```
void assign( size_type num, const TYPE &val );
```

//或者赋num个值为val的元素到vector中.这个函数将会清除掉为vector赋值以前的内容.

## at函数

### 语法:

TYPE at( size\_type loc ); //返回当前Vector指定位置loc的元素的引用.

at() 函数 比 [] 运算符更加安全, 因为它不会让你去访问到Vector内越界的元素. 例如, 考虑下面的代码:

```
vector v( 5, 1 );  
  
for( int i = 0; i < 10; i++ )  
{  
    cout << "Element " << i << " is " << v[i] << endl;  
}
```

这段代码访问了vector末尾以后的元素,这将可能导致很危险的结果.以下的代码将更加安全:

```
vector v( 5, 1 );  
  
for( int i = 0; i < 10; i++ )  
{  
    cout << "Element " << i << " is " << v.at(i) << endl;  
}
```

取代试图访问内存里非法值的作法,at() 函数能够辨别出访问是否越界并在越界的时候抛出一个异常.

## back 函数, front函数

### 语法:

TYPE back(); //back() 函数返回当前vector最末一个元素的引用.

TYPE front(); // front()函数返回当前vector起始元素的引用

例如:

```
vector v;  
  
for( int i = 0; i < 5; i++ )  
{  
    v.push_back(i);  
}  
  
cout <<"The first element is "<< v.front() <<" and the last element is" << v.back() << endl;
```

### 结果:

The first element is 0 and the last element is 4

## begin 函数 end 函数

### 语法:

iterator begin(); // begin()函数返回一个指向当前vector起始元素的迭代器.

iterator end(); // end() 函数返回一个指向当前vector末尾元素的下一位置的迭代器.注意,如果你要访问末尾元素,需要先将此迭代器自减1.

例如,下面这段使用了一个迭代器来显示出vector中的所有元素:

```
vector v1( 5, 789 ); vector::iterator it;
for( it = v1.begin(); it != v1.end(); it++ )

    cout << *it << endl;
```

### capacity 函数

**语法:**

size\_type capacity(); //capacity() 函数 返回当前vector在重新进行内存分配以前所能容纳的元素数量.

### clear 函数

**语法:**

```
void clear(); //clear()函数删除当前vector中的所有元素.

vector.clear();
```

### empty 函数

**语法:**

bool empty(); //如果当前vector没有容纳任何元素,则empty()函数返回true,否则返回false.

例如,以下代码清空一个vector,并按照**逆序显示**所有的元素:

```
vector v;

for( int i = 0; i < 5; i++ )
{
    v.push_back(i); //末尾添加元素
}

while( !v.empty() )
{
    cout << v.back() << endl;

    v.pop_back(); //删除末尾元素
}
```

### erase 函数

**语法:**

iterator erase( iterator loc ); // erase函数删作指定位置loc的元素

iterator erase( iterator start, iterator end ); //删除区间[start, end)的所有元素.返回值是指向删除的最后一个元素的下一位置的迭代器



例如:

// 创建一个vector,置入字母表的前十个字符

```
vector alphaVector;
```

```
for( int i=0; i < 10; i++ )
```

```
    alphaVector.push_back( i + 65 );
```

```
int size = alphaVector.size();
```

```
vector::iterator startIterator;
```

```
vector::iterator templIterator;
```

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

```
{
```

```
    startIterator = alphaVector.begin();
```

```
    alphaVector.erase( startIterator );
```

```
    // Display the vector
```

```
    for( templIterator = alphaVector.begin(); templIterator != alphaVector.end(); templIterator++ ) cout <<
        *templIterator; cout << endl;
```

```
}
```

**输出:**

BCDEFGHIJ CDEFGHIJ DEFGHIJ EFGHIJ FGHIJ GHIJ HIJ IJ J

### **get\_allocator 函数**

**语法:**

```
allocator_type get_allocator();
```

get\_allocator() 函数返回当前vector的内存分配器.

### **insert 函数**

**语法:**

insert() 函数有以下三种用法:

//在指定位置loc前插入值为val的元素,返回指向这个元素的迭代器,

```
iterator insert( iterator loc, const TYPE &val );
```

// 在指定位置loc前插入num个值为val的元素

```
void insert( iterator loc, size_type num, const TYPE &val );
```

// 在指定位置loc前插入区间[start, end)的所有元素 .

```
void insert( iterator loc, input_iterator start, input_iterator end );
```

举例:

```

//创建一个vector,置入字母表的前十个字符
vector alphaVector;

for( int i=0; i < 10; i++ )

    alphaVector.push_back( i + 65 );

//插入四个C到vector中

vector::iterator thelterator = alphaVector.begin();

alphaVector.insert( thelterator, 4, 'C' );

//显示vector的内容

for( thelterator = alphaVector.begin(); thelterator != alphaVector.end(); thelterator++ )

    cout << *thelterator;

```

**显示:**

CCCCABCDEFGHJ

### **max\_size 函数**

**语法:**

size\_type max\_size(); // 函数返回当前vector所能容纳元素数量的最大值(译注:包括可重新分配内存).

### **rbegin 函数**

**语法:**

reverse\_iterator rbegin();

//rbegin函数返回指向当前vector末尾的**逆迭代器**.(译注:实际指向末尾的下一位置,而其内容为末尾元素的值,详见逆代器相关内容)

### **rend 函数**

**语法:**

reverse\_iterator rend(); //rend()函数返回指向当前vector起始位置的**逆迭代器**.

### **reserve 函数**

**语法:**

void reserve( size\_type size );

//函数为当前vector预留至少共容纳size个元素的空间.(译注:实际空间可能大于size)

### **resize 函数**

**语法:**

void **resize**( size\_type size, TYPE val );

//resize() 函数改变当前vector的大小为size,且对新创建的元素赋值val

### **size 函数**

**语法:**

size\_type size(); //size() 函数返回当前vector所容纳元素的数目

**swap 函数**

**语法:**

void swap( vector &from ); //swap()函数交换当前vector与vector from的元素

---

应用

用vector存储二维数组

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

int main()
{
    int a[3][3]={1,2,3,4,5,6,7,8,9};
    vector<int> v(a[0],a[3]); //二维数组其实存储也是线性的
    vector<int>::iterator it;
    int item=5; //假如删除5
    for(it=v.begin();it!=v.end();it++)
    {
        if(* it==item)
            v.erase(it);
    }
    copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
    return 0;
}
```

---

来源网址:

[http://blog.sina.com.cn/s/blog\\_933dc4350100x9pz.html](http://blog.sina.com.cn/s/blog_933dc4350100x9pz.html)

<http://blog.csdn.net/u010585135/article/details/44995079>

---

## ACM/ICPC 竞赛之 STL--iterator

---

iterator(迭代器)是用于访问容器中元素的指示器，从这个意义上说，iterator(迭代器)相当于数据结构中所说的“遍历指针”，也可以把iterator(迭代器)看作是一种泛化的指针。STL 中关于iterator(迭代器)的实现是相当复杂的，这里我们暂时不去详细讨论关于iterator(迭代器)的实现和使用，而只对iterator(迭代器)做一点简单的介绍。

简单地说，STL 中有以下几类iterator(迭代器)：

输入iterator(迭代器)，在容器的连续区间内向前移动，可以读取容器内任意值；输出iterator(迭代器)，把值写进它所指向的容器中；前向iterator(迭代器)，读取队列中的值，并可以向前移动到下一位置(++p,p++)；双向iterator(迭代器)，读取队列中的值，并可以向前向后遍历容器；随机访问iterator(迭代器)，可以直接以下标方式对容器进行访问，vector 的iterator(迭代器)就是这种iterator(迭代器)；流iterator(迭代器)，可以直接输出、输入流中的值；每种STL 容器都有自己的iterator(迭代器)子类，下面先来看一段简单的示例代码：

```
#include <iostream>
#include <vector>
using namespace std;
main() {
    vector<int> s;
    for (int i=0; i<10; i++) s.push_back(i);
    for (vector<int>::iterator it=s.begin(); it!=s.end(); it++)
        cout << *it << " ";
    cout << endl;
    return 1;
}
```

vector 的begin()和end()方法都会返回一个vector::iterator 对象，分别指向vector 的首元素位置和尾元素的下一个位置（我们可以称之为结束标志位置）。对一个iterator(迭代器)对象的使用与一个指针变量的使用极为相似，或者可以这样说，指针就是一个非常标准的iterator(迭代器)。再来看一段稍微特别一点的代码：

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

main() {

    vector<int> s;

    s.push_back(1);

    s.push_back(2);

    s.push_back(3);

    copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));

    cout << endl;

    return 1;
}
```

这段代码中的copy 就是STL 中定义的一个模板函数, copy(s.begin(),s.end(), ostream\_iterator(cout, " "));的意思是由s.begin()至s.end()(不含s.end())所指定的序列复制到标准输出流cout 中, 用" 作为每个元素的间隔。也就是说, 这句话的作用其实就是将表中的所有内容依次输出。iterator(迭代器)是STL 容器和算法之间的“胶合剂”, 几乎所有的STL 算法都是通过容器的iterator(迭代器)来访问容器内容的。只有通过有效地运用iterator(迭代器), 才能够有效地运用STL 强大的算法功能。

## ACM/ICPC 竞赛之 STL--string

### 简介

字符串是程序中经常要表达和处理的数据, 我们通常是采用字符数组或字符指针来表示字符串。

STL为我们提供了另一种使用起来更为便捷的字符串的表达方式: string。string类的定义在头文件中。string类其实可以看作是一个字符的vector, vector上的各种操作都可以适用于string, 另外, string类对象还支持字符串的拼合、转换等操作。

### 一, 构造函数(转自参考网址)

a) string s; //生成一个空字符串 b) string s(str) //拷贝构造函数 生成str的复制品 c) string s(str,stridx) //将字符串str内“始于位置stridx”的部分当作字符串的初值 d) string s(str,stridx,strlen) //将字符串str内“始于stridx且长度顶多strlen”的部分作为字符串的初值 e) string s(cstr) //将C字符串作为s的初值 f) string s(chars,chars\_len) //将C字符串前chars\_len个字符作为字符串s的初值。 g) string s(num,c) //生成一个字符串, 包含num个c字符 h) string s(begin,end) //以区间begin;end(不包含end)内的字符作为字符串s的初值

### 二, 相关操作

1. 字符串操作函数 这里是C++字符串的重点, 我先把各种操作函数罗列出来, 不喜欢把所有函数都看完的人可以在这里找自己喜欢的函数, 再到后面看他的详细解释。 a) =, assign() //赋以新值 b) swap() //交换两个字符串的内容 c) +=, append(), push\_back() //在尾部添加字符 d) insert() //插入字符 e) erase() //删除字符 f) clear() //删除全部字符 g) replace() //替换字符 h) + //串联字符串 i) ==, !=, <, <=, >, >=, compare() //比较字符串 j) size(), length() //返回字符数量 k) max\_size() //返回字符的可能最大个数 l) empty() //判断字符串是否为空 m) capacity() //返回重新分配之前的字符容量 n) reserve() //保留一定量内存以容纳一定数量的字符 o) [ ], at() //存取单一字符 p) >>, getline() //从stream读取某值 q) << //将某值写入stream r) copy() //将某值赋值为一个C\_string s) c\_str() //将内容以C\_string返回 t) data() //将内容以字符数组形式返回 u) substr() //返回某个子字符串 v) 查找函数 w) begin() end() //提供类似STL的迭代器支持 x) rbegin() rend() //逆向迭代器 y) get\_allocator() //返回配置器 下面详细介绍:

#### 2.1 C++字符串和C字符串的转换

C++提供的由C++字符串得到对应的C\_string的方法是使用data()、c\_str()和copy(), 其中, data()以字符数组的形式返回字符串内容, 但并不添加'\0'。c\_str()返回一个以'\0'结尾的字符数组, 而copy()则把字符串的内容复制或写入既有c\_string或字符数组内。C++字符串并不以'\0'结尾。我的建议是在程序中能使用C++字符串就使用, 除非万不得已不选c\_string。

## 2. 2 大小和容量函数一个C++字符串存在三种大小:

a)现有的字符数, 函数是size()和length(), 他们等效。Empty()用来检查字符串是否为空。

b)max\_size() 这个大小是指当前C++字符串最多能包含的字符数, 很可能和机器本身的限制或者字符串所在位置连续内存的大小有关系。我们一般情况下不用关心他, 应该大小足够我们用的。但是不够用的话, 会抛出length\_error异常c)capacity()重新分配内存之前string所能包含的最大字符数。这里另一个需要指出的是reserve()函数, 这个函数为string重新分配内存。重新分配的大小由其参数决定, 默认参数为0, 这时候会对string进行非强制性缩减。还有必要再重复一下C++字符串和C字符串转换的问题, 许多人会遇到这样的问题, 自己做的程序要调用别人的函数、类什么的 (比如数据库连接函数Connect(char\*,char\*)), 但别人的函数参数用的是char\*形式的, 而我们知道, c\_str()、data()返回的字符数组由该字符串拥有, 所以是一种const char\*,要想作为上面提及的函数的参数, 还必须拷贝到一个char\*,而我们的原则是能不使用C字符串就不使用。那么, 这时候我们的处理方式是: 如果 此函数对参数(也就是char\*)的内容不修改的话, 我们可以这样Connect((char)UserID.c\_str(), (char\*)PassWD.c\_str()),但是这时候是存在危险的, 因为这样转换后的字符串其实是可以修改的 (有兴趣地可以自己试一试), 所以我强调除非函数调用的时候不对参数进行修改, 否则必须拷贝到一个char\*上去。当然, 更稳妥的办法是无论什么情况都拷贝到一个char\*上去。同时我们也祈祷现在仍然使用C字符串进行编程的高手们 (说他们是高手一点儿也不为过, 也许在我们还穿开裆裤的时候他们就开始编程了, 哈哈...) 写的函数都比较规范, 那样我们就不必进行强制转换了。

2. 3元素存取 我们可以使用下标操作符[]和函数at()对元素包含的字符进行访问。但是应该注意的是操作符[]并不检查索引是否有效 (有效索引0~str.length()), 如果索引失效, 会引起未定义的行为。而at()会检查, 如果使用at()的时候索引无效, 会抛出out\_of\_range异常。有一个例外不得不说, const string a;的操作符[]对索引值是a.length()仍然有效, 其返回值是'\0'。其他的各种情况, a.length()索引都是无效的。举例如下:

```
const string Cstr("const string"); string Str("string"); Str[3]; //ok Str.at(3); //ok Str[100]; //未定义的行为
Str.at(100); //throw out_of_range Str[Str.length()] //未定义行为 Cstr[Cstr.length()] //返回 '\0'
Str.at(Str.length()); //throw out_of_range Cstr.at(Cstr.length()) //throw out_of_range 我不赞成类似于下面的引用或指针赋值: char& r=s[2]; char* p= &s[3]; 因为一旦发生重新分配, r,p立即失效。避免的方法就是不使用。
```

2. 4比较函数 C++字符串支持常见的比较操作符 (>,>=,<,<=,==,!=), 甚至支持string与C-string的比较(如str<"hello")。在使用>,>=,<,<=这些操作符的时候是根据"当前字符特性"将字符按字典顺序进行逐一得比较。字典排序靠前的字符小, 比较的顺序是从前向后比较, 遇到不相等的字符就按这个位置上的两个字符的比较结果确定两个字符串的大小。同时, string("aaaa") 另一个功能强大的比较函数是成员函数compare()。他支持多参数处理, 支持用索引值和长度定位子串来进行比较。他返回一个整数来表示比较结果, 返回值意义如下: 0-相等 大于零0-大于, 小于0-小于。举例如下:

```
string s("abcd"); s.compare("abcd"); //返回0 s.compare("dcb"); //返回一个小于0的值 s.compare("ab"); //返回
大于0的值 s.compare(s); //相等 s.compare(0,2,s,2,2); //用"ab"和"cd"进行比较 小于零 s.compare(1,2,"bcx",2); //
用"bc"和"bc"比较。怎么样? 功能够全的吧! 什么? 还不能满足你的胃口? 好吧, 那等着, 后面有更个性化的比较
算法。先给个提示, 使用的是STL的比较算法。什么? 对STL一窍不通? 靠, 你重修吧!
```

2. 5 更改内容 这在字符串的操作中占了很大一部分。首先讲赋值, 第一个赋值方法当然是使用操作符=, 新值可以是string(如: s=ns)、c\_string(如: s="gaint")甚至单一字符 (如: s='j')。还可以使用成员函数assign(), 这个成员函数可以使你更灵活的对字符串赋值。还是举例说明吧:

```
s.assign(str); //不说 s.assign(str,1,3);//如果str是"iamangel" 就是把"ama"赋给字符串
s.assign(str,2,string::npos);//把字符串str从索引值2开始到结尾赋给 s s.assign("gaint"); //不说
s.assign("nico",5);//把'n' 'i' 'c' 'o' '\0'赋给字符串 s.assign(5,'x');//把五个x赋给字符串 把字符串清空的方法有三个:
s="";s.clear();s.erase();(我越来越觉得举例比说话让别人容易懂!)。string提供了很多函数用于插入 (insert)、
删除 (erase)、替换 (replace)、增加字符。先说增加字符 (这里说的增加是在尾巴上), 函数有 +=、
append()、push_back()。举例如下: s+=str;//加个字符串 s+="my name is jiayp";//加个C字符串 s+='a';//加个字
符 s.append(str); s.append(str,1,3);//不解释了 同前面的函数参数assign的解释 s.append(str,2,string::npos)//不
```

解释了 `s.append("my name is jiayp"); s.append("nico",5); s.append(5,'x'); s.push_back('a');` //这个函数只能增加单个字符 对STL熟悉的理解起来很简单也许你需要在string中间的某个位置插入字符串，这时候你可以用`insert()`函数，这个函数需要你指定一个安插位置的索引，被插入的字符串将放在这个索引的后面。 `s.insert(0,"my name"); s.insert(1,str);` 这种形式的`insert()`函数不支持传入单个字符，这时的单个字符必须写成字符串形式(让人恶心)。既然你觉得恶心，那就不得不继续读下面一段话：为了插入单个字符，`insert()`函数提供了两个对插入单个字符操作的重载函数：`insert(size_type index,size_type num,chart c)`和`insert(iterator pos,size_type num,chart c)`。其中 `size_type`是无符号整数，`iterator`是`char*`，所以，你这么调用`insert`函数是不行的：`insert(0,1,'j');`这时候第一个参数将转换成哪一个呢？所以你必须这么写：`insert((string::size_type)0,1,'j')!` 第二种形式指出了使用迭代器安插字符的形式，在后面会提及。顺便提一下，string有很多操作是使用STL的迭代器的，他也尽量做得和STL靠近。删除函数`erase()`的形式也有好几种（真烦！），替换函数`replace()`也有好几个。举例吧：`string s="il8n"; s.replace(1,2,"nternationalizatio");` //从索引1开始的2个替换成后面的C\_string

`s.erase(13);` //从索引13开始往后全删除 `s.erase(7,5);` //从索引7开始往后删5个

2. 6提取子串和字符串连接 题取子串的函数是：`substr()`，形式如下：`s.substr();` //返回s的全部内容 `s.substr(11);` //从索引11往后的子串 `s.substr(5,6);` //从索引5开始6个字符 把两个字符串结合起来的函数是`+`。（谁不明白请致电120） 2. 7输入输出操作 1. `>>` 从输入流读取一个string。 2. `<<` 把一个string写入输出流。另一个函数就是`getline()`，他从输入流读取一行内容，直到遇到分行符或到了文件尾。 2. 8搜索与查找 查找函数很多，功能也很强大，包括了：`find('lalala', 2)`

在str中，从位置2（包括位置2）开始，查找字符'lalala'，返回首次匹配的位置，若匹配失败，返回npos

`rfind()` `find_first_of()` `find_last_of()` `find_first_not_of()` `find_last_not_of()` 这些函数返回符合搜索条件的字符区间内的第一个字符的索引，没找到目标就返回npos。所有的函数的参数说明如下：第一个参数是被搜寻的对象。第二个参数（可有可无）指出string内的搜寻起点索引，第三个参数（可有可无）指出搜寻的字符个数。比较简单，不多说不理解的可以向我提出，我再仔细的解答。当然，更加强大的STL搜寻在后面会有提及。

最后再说说npos的含义，`string::npos`的类型是`string::size_type`，所以，一旦需要把一个索引与npos相比，这个索引值必须是`string::size_type`类型的，更多的情况下，我们可以直接把函数和npos进行比较（如：`if(s.find("jia")==string::npos)`）。

网址：

[http://blog.csdn.net/lina\\_acm/article/details/51474030](http://blog.csdn.net/lina_acm/article/details/51474030)

## ACM/ICPC 竞赛之 STL--list

### 一，构造函数

`list c`:创建一个空的list

`list c1(c2)`:复制另一个同类型元素的list

`listc(n)`:创建n个元素的list，每个元素值由默认构造函数确定

`listc(n,elem)`:创建n个元素的list，每个元素的值为elem

listc(begin,end):由迭代器创建list,迭代区间为[begin,end)

---

应用:

```
list<int>lst1;           //创建空list
list<int> lst2(5);       //创建含有5个元素的list
list<int>lst3(3,2);      //创建含有3个元素的list, 每个元素的值为2
list<int>lst4(lst2);     //使用lst2初始化lst4
list<int>lst5(lst2.begin(),lst2.end()); //同lst4
```

---

## 二, 相关操作



<b>list的非变动性操作</b>	
c.size()	返回容器的大小
c.empty()	判断容器是否为空,等价于size()==0,但可能更快
c.max_size()	返回容器最大的可以存储的元素
reserve()	如果容量不足, 扩大之
c1 == c2	判断c1 是否等于c2
c1 != c2	判断c1是否不等于c2
c1 < c2	判断c1 是否小于c2
c1 > c2	判断c1 是否大于c2
c1 <= c2	判断c1是否小于等于c2
c1 >= c2	判断c1是否大于等于c2
<b>list的赋值操作</b>	
c1 = c2	将c2的全部元素赋值给c1
c.assign(n, elem)	复制n个elem, 复制给c
c.assign(beg, end)	将区间[beg;end) 内的元素赋值给c
c1.swap(c2)	将c1和c2元素互换
swap(c1,c2)	同上, 此为全局函数
<b>list元素之间存取</b>	
c.front	返回第一个元素, 不检查元素存在与否
c.back	返回最后一个元素, 不检查元素存在与否
<b>list迭代器相关函数</b>	
c.begin()	返回一个双向迭代器, 指向第一个元素
c.end()	返回一个双向迭代器, 指向最后一个元素的下一个位置
c.rbegin()	返回一个逆向迭代器, 指向逆向迭代的第一个元素
c.rend()	返回一个逆向迭代器, 指向逆向迭代的最后一个元素的下一个位置
<b>list安插、移除操作函数</b>	
c.insert(pos, elem)	在迭代器pos所指位置上安插一个elem副本, 并返回新元素的位置

<b>list的非变动性操作</b>	
c.insert(pos,n,elem)	在pos位置上插入n个elem副本，无返回值
c.insert(pos,begin,end)	在pos位置上插入区间[begin,end)内的所有元素的副本没有返回值
c.push_back(elem)	在尾部添加一个elem副本
c.pop_back()	移除最后一个元素，无返回值
c.push_front()	在头部添加一个elem副本
c.pop_front()	移除第一个元素，但不返回
c.remove(val)	移除所有其值为val的元素
c.remove_if()	
c.erase(pos)	移除pos位置上的元素，返回下一个元素的位置
c.erase(begin, end)	移除(begin, end)区间内的所有元素，返回下一个元素的位置
c.resize(num)	将元素数量改为num(如果size()变大了，多出来的新元素都需以default构造函数完成)
c.resize(num,elem)	将元素数量改为num(如果size()变大了，多出来的新元素都是elem的副本)
c.clear()	移除所有元素，将容器清空
备注：安插和移除元素，都会使“作用点”之后的各个元素的iterator等失效，若发生内存重新分配，该容器身上的所有iterator等都会失效	
List的特殊变动性操作	
c.unique()	如果存在若干相邻而数值相等的元素，就移除重复元素，只留下一个
c.unique(op)	如果存在若干相邻元素，都使op()的结果为ture，则移除重复元素，只留下一个。
c1.splice(pos, c2)	将c2内的所有元素转移到c1之内，迭代器pos之前
c1.splice(pos, c2, c2pos)	将c2内的c2pos所指元素转移到c1之内的pos所指位置上(c1,c2可相同)
c1.splice(pos, c2, c2beg, c2end)	将c2内的[c2beg, c2end) 区间内所有元素转移到c1内的pos之前(c1,c2可相同)
c.sort()	以operator<为准则，对所有元素排序
c.sort(op)	以op()为准则，对所有元素排序

list的非变动性操作	
c1.merge(c2)	假设c1和c2容器都包含已序(相同的排序方式)元素，将c2的全部元素转移到c1，并保证合并后的list还是已序。
c1.merge(c2, op)	假设c1和c2容器都包含op()原则下的已序(相同的排序方式)元素，将c2的全部元素转移到c1，并保证合并后的list在op () 原则仍是已序。
c.reverse()	将所有元素反

应用（转载自<http://blog.csdn.net/nuptboyzhb/article/details/8120397>）：

### (一)list简介：

list不像vector那样，list的内存分配时非连续的，因此，只能通过迭代器来访问list中的元素。另外，list在头和尾都可以插入元素

(二)创建一个list 1.首先包含list的头文件2.使用标准的std命名空间#include using namespace std;一下是构造list的不同方法：

[cpp] [view plain copy](#)

```
1. <span style="font-size:18px;">#pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <string>
5. using namespace std;
6. class Student{
7. private:
8.     int ID;
9.     string Name;
10. public:
11.     Student(int ID,string Name)
12.     {
13.         this->ID=ID;
14.         this->Name=Name;
15.     }
16.     int getID()
17.     {
18.         return ID;
19.     }
20.     string getName()
21.     {
22.         return Name;
23.     }
24. };
25. int main()
```

```

26. {
27.     // create an empty list (of zero size) capable of holding doubles
28.     list<double> list0;
29.
30.     cout << "Size of list0 is " << list0.size() << endl;
31.
32.     // create a list with 5 empty elements
33.     list<double> list1(5);
34.
35.     cout << "Size of list1 is " << list1.size() << endl;
36.
37.     // create a list with 5 elements, each element having the value 10.2
38.     list<double> list2(5, 10.2);
39.
40.     cout << "list2: ";
41.     list<double>::iterator it;
42.     for(it = list2.begin(); it != list2.end(); ++it)
43.         cout << *it << " ";
44.     cout << endl;
45.     // create a list based on an array of elements
46.     // only the first 5 elements of the array are copied into the vector
47.     double array[8] = {3.45, 67, 10, 0.67, 8.99, 9.78, 6.77, 34.677};
48.
49.     list<double> list3(array, array + 5);
50.
51.     cout << "list3: ";
52.     for(it = list3.begin(); it != list3.end(); ++it)
53.         cout << *it << " ";
54.     cout << endl;
55.
56.     // use the copy constructor to copy list3 list into list3copy list
57.     list<double> list3copy(list3);
58.
59.     cout << "list3copy: ";
60.     for(it = list3copy.begin(); it != list3copy.end(); ++it)
61.         cout << *it << " ";
62.     cout << endl;
63.
64.     // assign 5 values of 10.2 to the list
65.     list<double> list4;
66.
67.     list4.assign(5, 10.2);
68.
69.     cout << "list4: ";
70.     for(it = list4.begin(); it != list4.end(); ++it)
71.         cout << *it << " ";
72.     cout << endl;
73.     //定义自己的数据类型
74.     list<Student> list5;
75.     Student stu1(1, "ZhengHaibo");
76.     Student stu2(2, "nupt");
77.     list5.push_back(stu1);
78.     list5.push_back(stu2);

```

```

79.     list<Student>::iterator iter_stu;
80.     cout << "list5: "<<endl;
81.     for (iter_stu=list5.begin();iter_stu!=list5.end();iter_stu++)
82.     {
83.         cout<<"ID:"<<iter_stu-getID()<<" Name:"<<iter_stu->getName()<<endl;
84.     }
85.     return 0;
86.     // Output
87.     // Size of list0 is 0
88.     // Size of list1 is 5
89.     // list2: 10.2 10.2 10.2 10.2 10.2
90.     // list3: 3.45 67 10 0.67 8.99
91.     // list3copy: 3.45 67 10 0.67 8.99
92.     // list4: 10.2 10.2 10.2 10.2 10.2
93.     //list5:
94.     //ID:1 Name:ZhengHaibo
95.     //ID:2 Name:nupt
96. }
97. </span>

```

**(三)访问输出list中的值** 在list中，由于其内存是非连续的，因此不能像vector那样，用[]操作符取值，只能用迭代器。

[cpp] [view plain copy](#)

```

1. <span style="font-size:18px;">#pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <string>
5. using namespace std;
6. //Printing the elements of a list can only be done through iterators.
7. void print(list<double> lst, char * name)
8. {
9.     list<double>::iterator it;
10.    cout << name << ": ";
11.    for(it = lst.begin(); it != lst.end(); ++it)
12.        cout << *it << " ";
13.    cout << endl;
14. }
15. int main()
16. {
17.    double array[8] = {3.45, 67, 10, 0.67, 8.99, 9.78, 6.77, 34.677};
18.    list<double> lst(array, array + 5);
19.    print(lst, "lst");
20.    cout << "lst in reverse order: ";
21.    list<double>::reverse_iterator rit;//
22.    for(rit = lst.rbegin(); rit != lst.rend(); ++rit)
23.        cout << *rit << " ";
24.    return 0;
25.    //Output

```

```
26. // lst: 3.45 67 10 0.67 8.99
27. // lst in reverse order: 8.99 0.67 10 67 3.45
28. }
```

**(四)向list中插入元素** 与vector相比, list除了有push\_back()//在尾部插入 和 insert()之外, 还有push\_front()//即在链表的头部插入

[cpp] [view plain copy](#)

```
1. <span style="font-size:18px;">#pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <string>
5. using namespace std;
6. //Printing the elements of a list can only be done through iterators.
7. void print(list<double> lst, char * name)
8. {
9.     list<double>::iterator it;
10.    cout << name << ": ";
11.    for(it = lst.begin(); it != lst.end(); ++it)
12.        cout << *it << " ";
13.    cout << endl;
14. }
15. int main()
16. {
17.    list<double> lst;
18.    list<double>::iterator it;
19.
20.    // add elements to the end of the list
21.    lst.push_back(2.4);
22.    lst.push_back(4.5);
23.    lst.push_back(0.98);
24.
25.    print(lst, "lst");
26.
27.    // insert value 6.7 in the second position in the list
28.    it = lst.begin();
29.    lst.insert(++it, 6.7);
30.
31.    print(lst, "lst");
32.
33.    // insert elements from the array at the end of the list
34.    double array[2] = {100.89, 789.76};
35.    it = lst.end();
36.    lst.insert(it, array, array + 2);
37.
38.    print(lst, "lst");
39.
40.    // add elements to the beginning of the list
41.    lst.push_front(0.45);
42.    lst.push_front(0.56);
43.    lst.push_front(0.78);
44.
```

```

45.     print(lst, "lst");
46.     return 0;
47.     // Output
48.     // lst: 2.4 4.5 0.98
49.     // lst: 2.4 6.7 4.5 0.98
50.     // lst: 2.4 6.7 4.5 0.98 100.89 789.76
51.     // lst: 0.78 0.56 0.45 2.4 6.7 4.5 0.98 100.89 789.76
52. }</span>

```

**(五)删除list中的元素** 主要有如下函数: pop\_back(),pop\_front(),erase(),remove(),remove\_if(),unique(),clear(); 与插入元素相对应, list用pop\_back()//在尾部删除一个元素,pop\_front()//在头部删除一个元素,erase()删除任意部位

[cpp] [view plain copy](#)

```

1. <span style="font-size:18px;">#pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <string>
5. using namespace std;
6. //Printing the elements of a list can only be done through iterators.
7. void print(list<double> lst, char * name)
8. {
9.     list<double>::iterator it;
10.    cout << name << ": ";
11.    for(it = lst.begin(); it != lst.end(); ++it)
12.        cout << *it << " ";
13.    cout << endl;
14. }
15. int main()
16. {
17.    double array[10] = {3.45, 67, 10, 0.67, 8.99, 9.78, 6.77, 34.677, 100.67, 0.99};
18.    list<double> lst(array, array + 10);
19.    list<double>::iterator it;
20.
21.    print(lst, "lst");
22.
23.    // remove the last element of the list
24.    lst.pop_back();
25.
26.    print(lst, "lst");
27.
28.    // remove the second element of the list
29.    it = lst.begin();
30.    lst.erase(++it);
31.
32.    print(lst, "lst");
33.
34.    // remove the first element of the list
35.    lst.pop_front();
36.
37.    print(lst, "lst");
38.    return 0;
39.    // Output

```

```

40.    // lst: 3.45 67 10 0.67 8.99 9.78 6.77 34.677 100.67 0.99
41.    // lst: 3.45 67 10 0.67 8.99 9.78 6.77 34.677 100.67
42.    // lst: 3.45 10 0.67 8.99 9.78 6.77 34.677 100.67
43.    // lst: 10 0.67 8.99 9.78 6.77 34.677 100.67
44. }</span>

```

另外，list还有remove()函数，remove()函数也是删除一个元素，但是，它的参数是元素的值或者是对象，而不是迭代器。同时，还有remove\_if()函数，该函数的参数是一个函数，是一个返回值为bool，参数为元素类型的一个函数。

//注意：该程序在VC6.0下编译报错，建议使用g++或gcc。

[cpp] [view plain copy](#)

```

1. <span style="font-size:18px;">#pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <string>
5. using namespace std;
6. //Printing the elements of a list can only be done through iterators.</span>

```

[cpp] [view plain copy](#)

```

<span style="font-size:18px;">////http://blog.csdn.net/nuptboyzhb/article/details/8120397

```

[cpp] [view plain copy](#)

```

1. <span style="font-size:18px;">
2. void print(list<double> lst, char * name)
3. {
4.     list<double>::iterator it;
5.     cout << name << ": ";
6.     for(it = lst.begin(); it != lst.end(); ++it)
7.         cout << *it << " ";
8.     cout << endl;
9. }
10. bool zhb_predicate(double& element)
11. {
12.     return (element >= 15.0) ? true : false;
13. }
14. int main()
15. {
16.     double array[10] = {3.45, 67, 19.25, 0.67, 8.99, 9.78, 19.25, 34.677, 100.67, 19.25};
17.     list<double> lst(array, array + 10);
18.

```



```

19.     print(lst, "lst");
20.
21.     // remove all elements with value 19.25 from the list
22.     lst.remove(19.25);
23.
24.     print(lst, "lst");
25.
26.     lst.remove_if(zhb_predicate);
27.
28.     print(lst, "lst");
29.     return 0;
30.     //lst: 3.45 67 19.25 0.67 8.99 9.78 19.25 34.677 100.67 19.25
31.     //lst: 3.45 67 0.67 8.99 9.78 34.677 100.67
32.     //lst: 3.45 0.67 8.99 9.78
33. }</span>

```

另外一个删除元素的函数是unique()函数，该函数有2种调用方式。无参数调用的情况下，如果该元素与其前一个元素相同，则删除之。当然，我们也可以传递一个判断是否相等的一个函数。

[c++ code list6.cpp]//注意：该程序在VC6.0下编译报错，建议使用g++或gcc

[cpp] [view plain copy](#)

```

1. <span style="font-size:18px;">#pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <cmath>
5. #include <string>
6. using namespace std;
7. //Printing the elements of a list can only be done through iterators.
8. void print(list<double> lst, char * name)
9. {
10.     list<double>::iterator it;
11.     cout << name << ": ";
12.     for(it = lst.begin(); it != lst.end(); ++it)
13.         cout << *it << " ";
14.     cout << endl;
15. }
16. bool almost_equal(double& e1, double& e2)
17. {
18.     return (fabs(e1 - e2) <= 0.1) ? true : false;
19. }
20. int main()
21. {
22.     double array[10] = {3.45, 0.67, 0.67, 0.62, 8.99, 8.98, 8.99, 34.677, 100.67, 19.25};
23.     list<double> lst(array, array + 10);
24.     print(lst, "lst");
25.     // remove all duplicate elements from the list
26.     lst.unique();
27.     print(lst, "lst");
28.     // remove all duplicate elements from the list

```

```

29.     lst.unique(almost_equal);
30.
31.     print(lst, "lst");
32.
33.     return 0;
34.     //Output
35.     // lst: 3.45 0.67 0.67 0.62 8.99 8.98 8.99 34.677 100.67 19.25
36.     // lst: 3.45 0.67 0.62 8.99 8.98 8.99 34.677 100.67 19.25
37.     // lst: 3.45 0.67 8.99 34.677 100.67 19.25
38. }</span>

```

**(六)链表的长度及重置链表的长度** 与vector不同的是，list没有capacity() 和 reserve()函数。list用size获得链表的长度，用resize改变其大小。

[cpp] [view plain copy](#)

```

1. <span style="font-size:18px;">#pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <cmath>
5. #include <string>
6. using namespace std;
7. //Printing the elements of a list can only be done through iterators.
8. void print(list<double> lst, char * name)
9. {
10.     list<double>::iterator it;
11.     cout << name << ": ";
12.     for(it = lst.begin(); it != lst.end(); ++it)
13.         cout << *it << " ";
14.     cout << endl;
15. }
16. int main()
17. {
18.     list<double> lst;
19.
20.     lst.push_back(0.67);
21.     lst.push_back(7.89);
22.     lst.push_back(3.56);
23.     lst.push_back(10.67);
24.     lst.push_back(9.89);
25.
26.     cout << "lst size is " << lst.size() << endl;
27.     print(lst, "lst");
28.
29.     // case when new size <= size of list
30.     lst.resize(3);
31.
32.     cout << "lst size is " << lst.size() << endl;
33.     print(lst, "lst");
34.
35.     // case when new size > size of list
36.     lst.resize(10);

```

```

37.
38.     cout << "lst size is " << lst.size() << endl;
39.     print(lst, "lst");
40.     return 0;
41.     // Output
42.     // lst size is 5
43.     // lst: 0.67 7.89 3.56 10.67 9.89
44.     // lst size is 3
45.     // lst: 0.67 7.89 3.56
46.     // lst size is 10
47.     // lst: 0.67 7.89 3.56 0 0 0 0 0 0
48. }</span>

```

**(七)反转一个链表** 调用reverse()函数即可，很简单，不再举例。**(八)链表元素的排序** 用sort()函数进行排序，默认情况下是从小到大，当然，也可以人为地改变。

//注意：该程序在VC6.0下编译报错，建议使用g++或gcc

[cpp] [view plain copy](#)

```

1. <span style="font-size:18px;">#pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <cmath>
5. #include <string>
6. using namespace std;
7. //Printing the elements of a list can only be done through iterators.
8. void print(list<double> lst, char * name)
9. {
10.     list<double>::iterator it;
11.     cout << name << ": ";
12.     for(it = lst.begin(); it != lst.end(); ++it)
13.         cout << *it << " ";
14.     cout << endl;
15. }
16. int main()
17. {
18.     double array[10] = {3.45, 3.455, 67, 0.67, 8.99, 9.0, 9.01, 34.677, 100.67, 19.25};
19.     list<double> lst(array, array + 10);
20.
21.     print(lst, "lst");
22.
23.     // sort the list;
24.     // < operator will be used as default
25.     // the elements will be sorted in ascending order
26.     lst.sort();
27.
28.     print(lst, "lst in ascending order");
29.
30.     // sort the list; specify the sorting function
31.     // > operator will be used in this case
32.     // the elements will be sorted in descending order

```

```

33.     lst.sort( greater<double>() );
34.
35.     print(lst, "lst in descending order");
36.
37.     // sort the list; specify the sorting function
38.     // < operator will be used in this case
39.     // the elements will be sorted in descending order
40.     lst.sort( less<double>() );
41.
42.     print(lst, "lst in ascending order");
43.     return 0;
44.     // Output
45.     // lst: 3.45 3.455 67 0.67 8.99 9 9.01 34.677 100.67 19.25
46.     // lst in ascending order: 0.67 3.45 3.455 8.99 9 9.01 19.25 34.677 67 100.67
47.     // lst in descending order: 100.67 67 34.677 19.25 9.01 9 8.99 3.455 3.45 0.67
48.     // lst in ascending order: 0.67 3.45 3.455 8.99 9 9.01 19.25 34.677 67 100.67
49. }</span>

```

**(九)交换两个链表** 直接调用swap()函数即可。lst1.swap(lst2); **(十)合并两个链表** 主要有两个函数：splice()和merge() splice()有三种调用形式： 第一种： list1.splice(it1, list2).将list2中的所有元素拷贝到list1中。在list1中的起始位置是it1.复制结束后，list2将为空。 [c++ code]

[cpp] [view plain copy](#)

```

1. list<double> list1;
2. list<double> list2;
3.
4. list1.push_back(1.0);
5. list1.push_back(5.0);
6. list1.push_back(6.0);
7. list1.push_back(7.0);
8. list1.push_back(8.0);
9.
10. list2.push_back(2.0);
11. list2.push_back(3.0);
12. list2.push_back(4.0);
13.
14. cout << "Before splice: " << endl;
15. print(list1, "list1");
16. print(list2, "list2");
17.
18. list<double>::iterator it1 = list1.begin();
19. ++it1;
20.
21. // move all the elements of list2 into list1,
22. // starting from the second position
23. list1.splice(it1, list2);
24.
25. cout << "After splice: " << endl;
26. print(list1, "list1");
27. print(list2, "list2");
28.

```

```
29. // Output
30. // Before splice:
31. // list1: 1 5 6 7 8
32. // list2: 2 3 4
33. // After splice:
34. // list1: 1 2 3 4 5 6 7 8
35. // list2:
```

第二种调用形式：list1.splice(it1, list2, it2) 这个功能是将list2中的元素，从it2开始，剪切到list1的it1起始的地方。第三种调用形式：list1.splice(it1, list2, it2begin, it2end) merge函数的使用：形式：list1.merge(list2) 注意：在使用merge之前，必须使list1和list2已经排好顺序。并且，合并之后list1仍然是有序的。

```
list<double> list1;

list<double> list2;

list1.push_back(1.0);

list1.push_back(5.0);

list1.push_back(6.0);

list1.push_back(7.0);

list1.push_back(8.0);

list2.push_back(2.0);

list2.push_back(3.0);

list2.push_back(4.0);

cout << "Beforemerge: " <<endl;

print(list1, "list1");

print(list2, "list2");


// merge the two lists

list1.merge(list2);

cout << "Aftermerge: " <<endl;

print(list1, "list1");

print(list2, "list2");
```

```
// Output

// Before merge:

// list1: 1 5 6 7 8

// list2: 2 3 4

// After merge:

// list1: 1 2 3 4 5 6 7 8

// list2:
```

**(十一)二维链表** 所谓二维链表就是链表的链表。

```
list< list<double> > matrix;

list<double> lst1(5, 6.713);

list<double> lst2(6, 5.678);

matrix.push_back(lst1);

matrix.push_back(lst2);

list< list<double> >::iterator it2d;

for(it2d = matrix.begin(); it2d != matrix.end(); it2d++)

    print(*it2d, "row");

// Output

// row: 6.713 6.713 6.713 6.713 6.713

// row: 5.678 5.678 5.678 5.678 5.678 5.678
```

**(十二)用户自定义的元素类型** 为了能够使用stl中的算法，用户自定义的类必须实现很多运算符的重载。g++编译通过

**[cpp]** [view plain copy](#)

```
1. #pragma warning(disable:4786)
2. #include <iostream>
3. #include <list>
4. #include <cmath>
5. #include <string.h>
6. using namespace std;

7. class Person
```

```

8. {
9.     char * name;
10.    char sex;
11.    int age;
12.
13. public:
14.
15.    // constructor
16.    Person()
17.    {
18.        name = new char[strlen("Anonymous") + 1];
19.        sex = 'N';
20.        age = 0;
21.    }
22.
23.    // constructor
24.    Person(char * pName, char pSex, int pAge)
25.        : sex(pSex), age(pAge)
26.    {
27.        name = new char[strlen(pName) + 1];
28.        strcpy(name, pName);
29.    }
30.
31.    // copy constructor
32.    Person(const Person& rhs)
33.        : sex(rhs.sex), age(rhs.age)
34.    {
35.        name = new char[strlen(rhs.name) + 1];
36.        strcpy(name, rhs.name);
37.    }
38.
39.    // overload the assignment operator
40.    Person& operator=(const Person& rhs)
41.    {
42.        name = new char[strlen(rhs.name) + 1];
43.        strcpy(name, rhs.name);
44.        sex = rhs.sex;
45.        age = rhs.age;
46.
47.        return *this;
48.    }
49.
50.    // overload the == operator
51.    // for sorting purposes, we consider that two Person objects are "equal"
52.    // if they have the same age
53.    bool operator==(const Person& rhs) const
54.    {
55.        return (age == rhs.age) ? true : false;
56.    }
57.
58.    // overload the < operator
59.    // for sorting purposes, we consider that a Person object is "less than" another
60.    // if it's age is less than the other object's age

```

```

61.     bool operator<(const Person& rhs) const
62.     {
63.         return (age < rhs.age) ? true : false;
64.     }
65.
66.     // overload the > operator
67.     // for sorting purposes, we consider that a Person object is "greater than" another
68.     // if it's age is greater than the other object's age
69.     bool operator>(const Person& rhs) const
70.     {
71.         return (age > rhs.age) ? true : false;
72.     }
73.
74.     // print the object
75.     void print()
76.     {
77.         cout << name << " " << sex << " " << age << endl;
78.     }
79.
80.     // destructor
81.     ~Person()
82.     {
83.         delete []name;
84.     }
85. };
86. void print(list<Person> lst, char * name)
87. {
88.     list<Person>::iterator it;
89.
90.     cout << name << ":" << endl;
91.
92.     for(it = lst.begin(); it != lst.end(); ++it)
93.         it->print();
94.
95.     cout << endl;
96. }
97. int main()
98. {
99.     list<Person> lst;
100.
101.     // create some Person objects
102.     Person p1("Bill Gates", 'M', 50);
103.     Person p2("Scott Meyers", 'M', 43);
104.     Person p3("Charles Petzold", 'M', 48);
105.     Person p4("Christina Dermayr", 'F', 30);
106.     Person p5("Andrei Neagu", 'M', 22);
107.     Person p6("Yin Su", 'F', 56);
108.     Person p7("Georgeta Bills", 'F', 37);
109.
110.     // add the objects to the list
111.     lst.push_back(p1);
112.     lst.push_back(p2);
113.     lst.push_back(p3);

```



```

114.     lst.push_back(p4);
115.     lst.push_back(p5);
116.     lst.push_front(p6);
117.     lst.push_front(p7);
118.     print(lst, "lst");
119.     // sort the list in ascending order
120.     lst.sort( less<Person>() );
121.
122.     print(lst, "lst in ascending order");
123.
124.     // sort the list in descending order
125.     lst.sort( greater<Person>() );
126.
127.     print(lst, "lst in descending order");
128.
129.     // delete the first element from the list
130.     lst.pop_front();
131.
132.     print(lst, "lst");
133.
134.     // clear the list
135.     lst.clear();
136.
137.     if(lst.empty())
138.         cout << "lst is empty" << endl;
139.     else
140.         cout << "lst is not empty" << endl;
141.     return 0;
142.     // Output
143.     // lst:
144.     // Georgeta Bills F 37
145.     // Yin Su F 56
146.     // Bill Gates M 50
147.     // Scott Meyers M 43
148.     // Charles Petzold M 48
149.     // Christina Dermayr F 30
150.     // Andrei Neagu M 22
151.     //
152.     // lst in ascending order:
153.     // Andrei Neagu M 22
154.     // Christina Dermayr F 30
155.     // Georgeta Bills F 37
156.     // Scott Meyers M 43
157.     // Charles Petzold M 48
158.     // Bill Gates M 50
159.     // Yin Su F 56
160.     //
161.     // lst in descending order:
162.     // Yin Su F 56
163.     // Bill Gates M 50
164.     // Charles Petzold M 48
165.     // Scott Meyers M 43
166.     // Georgeta Bills F 37

```

```
167.    // Christina Dermayr F 30
168.    // Andrei Neagu M 22
169.    //
170.    // lst:
171.    // Bill Gates M 50
172.    // Charles Petzold M 48
173.    // Scott Meyers M 43
174.    // Georgeta Bills F 37
175.    // Christina Dermayr F 30
176.    // Andrei Neagu M 22
177.    //
178.    // lst is empty
179. }
```

来源网址:

[http://blog.csdn.net/c\\_base\\_jin/article/details/51318621](http://blog.csdn.net/c_base_jin/article/details/51318621)

<http://blog.csdn.net/longshengguoji/article/details/8520891>

<http://blog.csdn.net/nupt123456789/article/details/8120397>

<http://www.cnblogs.com/qi09/archive/2011/01/21/1941065.html>

## ACM/ICPC 竞赛之 STL--stack/queue

### 简介

stack(栈)和queue(队列)也是在程序设计中经常会用到的数据容器，STL为我们提供了方便的stack(栈)的queue(队列)的实现。准确地说，STL 中的stack 和queue 不同于vector、list 等容器，而是对这些容器的重新包装。这里我们不去深入讨论STL 的stack 和queue 的实现细节，而是来了解一些他们的基本使用。

### 一，stack:

#### 1，构造函数

stack () 生成一个空栈

#### 2，相关操作

s.push(x); 入栈

s.pop(); 出栈，注意，出栈操作只是删除栈顶元素，并不返回该元素。

s.top(); 访问栈顶元素

s.empty(), 判断栈空，当栈空时，返回true。

s.size() 访问栈中元素个数

应用：下面是用string 和stack 写的解题1064--Parencoding 的程序。

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int main(){
    int n;
    cin >> n;
    for (int i=0; i<n; i++){
        int m;
        cin >> m;
        string str;
        int leftpa = 0;
        for (int j=0; j<m; j++) { // 读入P编码, 构造括号字符串
            int p;
            cin >> p;
            for (int k=0; k<p-leftpa; k++) str += '(';
            str += ')';
            leftpa = p;
        }
        stack<int> s;
        for (string::iterator it=str.begin();
            it!=str.end(); it++) { // 构造M编码
            if (*it=='(') s.push(1);
            else{
                int p = s.top(); s.pop();
                cout << p << " ";
                if (!s.empty()) s.top() += p;
            }
        }
        cout << endl;
    }
    return 0;
}
```

## 二, queue:

### 1, 构造函数

queue() 创建一个空队列

### 2, 相关操作

q.push(x); 入队, 将x 接到队列的末端。

q.pop(); 出队, 弹出队列的第一个元素, 注意, 并不会返回被弹出元素的值。

q.front(), 访问队首元素, 即最早被压入队列的元素。

q.back(), 访问队尾元素, 即最后被压入队列的元素。

q.empty(), 判断队空, 当队列空时, 返回true。

q.size() 访问队中元素个数

---

应用: 约瑟夫问题

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
    queue<int> qu;
    for (int i = 1; i <= 39; i++) {
        qu.push(i);
    }

    while(!qu.empty()) {
        for(int i = 1; i < 3; i++) {
            int te = qu.front();
            qu.pop();
            qu.push(te);
        }
        cout << qu.front() << " ";
        qu.pop();
    }
    return 0;
}
```

---

## 三, priority\_queue:

### 简介:

优先队列是队列的一种, 不过它可以按照自定义的一种方式(数据的优先级)来对队列中的数据进行动态的排序每次的push和pop操作, 队列都会动态的调整, 以达到我们预期的方式来存储。

### 1, 构造函数

priority\_queue (); 创建一个空队列

### 2, 相关操作

q.empty() const; 检查优先级队列是否为空, 若为空, 返回TRUE, 否则返回ELASE

q.pop(); 从优先级队列中删除优先级最大的元素, 前提条件: 队列不为空

q.push(const T &item); 向优先级队列中插入一个元素

q.size() const; 返回优先级队列中元素的个数

T & top(); 返回具有最高优先级的元素的引用

const T & top() const; 常量版top ()

*关于优先级的定义请看应用*

---

应用 (转自<https://www.cnblogs.com/summerRQ/articles/2470130.html>)

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

//定义比较结构
struct cmp1{
    bool operator()(int &a,int &b){
        return a>b;//最小值优先
    }
};

struct cmp2{
    bool operator()(int &a,int &b){
        return a<b;//最大值优先
    }
};

//自定义数据结构
struct number1{
    int x;
    bool operator < (const number1 &a) const {
        return x>a.x;//最小值优先
    }
};

struct number2{
    int x;
    bool operator < (const number2 &a) const {
        return x<a.x;//最大值优先
    }
};

int a[]={14,10,56,7,83,22,36,91,3,47,72,0};
number1 num1[]={14,10,56,7,83,22,36,91,3,47,72,0};//语法同数组语法, 必须为公有变量
number2 num2[]={14,10,56,7,83,22,36,91,3,47,72,0};//

int main()
{
    priority_queue<int>que;//采用默认优先级构造队列

    //priority_queue<int,vector<int>,greater<int>>que1;//最小值优先
    // greater与less https://www.cnblogs.com/lzhu/p/7010894.html
```

```

priority_queue<int,vector<int>,cmp1>que1;//最小值优先
priority_queue<int,vector<int>,cmp2>que2;//最大值优先

priority_queue<int,vector<int>,greater<int> >que3;//注意“>”会被认为错误,
priority_queue<int,vector<int>,less<int> >que4;////最大值优先

priority_queue<number1>que5; //最小优先级队列
priority_queue<number2>que6; //最大优先级队列

int i;
for(i=0;a[i];i++){
    que.push(a[i]);
    que1.push(a[i]);
    que2.push(a[i]);
    que3.push(a[i]);
    que4.push(a[i]);
}
for(i=0;num1[i].x;i++)
    que5.push(num1[i]);
for(i=0;num2[i].x;i++)
    que6.push(num2[i]);

printf("采用默认优先关系:/n(priority_queue<int>que;)/n");
printf("Queue 0:/n");
while(!que.empty()){
    printf("%3d",que.top());
    que.pop();
}
puts("");
puts("");

printf("采用结构体自定义优先级方式一:/n(priority_queue<int,vector<int>,cmp>que;)/n");
printf("Queue 1:/n");
while(!que1.empty()){
    printf("%3d",que1.top());
    que1.pop();
}
puts("");
printf("Queue 2:/n");
while(!que2.empty()){
    printf("%3d",que2.top());
    que2.pop();
}
puts("");
puts("");
printf("采用头文件/functional/内定义优先
级:/n(priority_queue<int,vector<int>,greater<int>/less<int> >que;)/n");
printf("Queue 3:/n");
while(!que3.empty()){
    printf("%3d",que3.top());
    que3.pop();
}

```

```

    puts("");
    printf("Queue 4:/n");
    while(!que4.empty()){
        printf("%3d",que4.top());
        que4.pop();
    }
    puts("");
    puts("");
    printf("采用结构体自定义优先级方式二:/n(priority_queue<number>que)/n");
    printf("Queue 5:/n");
    while(!que5.empty()){
        printf("%3d",que5.top());
        que5.pop();
    }
    puts("");
    printf("Queue 6:/n");
    while(!que6.empty()){
        printf("%3d",que6.top());
        que6.pop();
    }
    puts("");
    return 0;
}
/*

```

运行结果：

采用默认优先关系：

```
(priority_queue<int>que;)
```

Queue 0:

```
91 83 72 56 47 36 22 14 10 7 3
```

采用结构体自定义优先级方式一：

```
(priority_queue<int,vector<int>,cmp>que;)
```

Queue 1:

```
3 7 10 14 22 36 47 56 72 83 91
```

Queue 2:

```
91 83 72 56 47 36 22 14 10 7 3
```

采用头文件"functional"内定义优先级：

```
(priority_queue<int,vector<int>,greater<int>/less<int> >que;)
```

Queue 3:

```
3 7 10 14 22 36 47 56 72 83 91
```

Queue 4:

```
91 83 72 56 47 36 22 14 10 7 3
```

采用结构体自定义优先级方式二：

```
(priority_queue<number>que)
```

Queue 5:

```
3 7 10 14 22 36 47 56 72 83 91
```

Queue 6:

```
91 83 72 56 47 36 22 14 10 7 3
```

```
*/
```

来源网址:

<https://www.cnblogs.com/icode-girl/p/4978837.html>

<https://www.cnblogs.com/summerRQ/articles/2470130.html>

greater与less <https://www.cnblogs.com/lzhu/p/7010894.html> 注意区分排序与优先级的不同

## ACM/ICPC 竞赛之 STL--set

### 简介:

关于set, 必须说明的是set关联式容器。set作为一个容器也是用来存储同一数据类型的数据类型, 并且能从一个数据集中取出数据, 在set中每个元素的值都唯一, 而且系统能根据元素的值自动进行排序。应该注意的是set中数元素的值不能被直接改变。C++ STL中标准关联容器set, multiset, map, multimap内部采用的就是一种非常高效的平衡检索二叉树: 红黑树, 也成为RB树(Red-Black Tree)。RB树的统计性能要好于一般平衡二叉树, 所以被STL选择作为关联容器的内部结构。

set/multiset会根据待定的排序准则, 自动将元素排序。两者不同在于前者不允许元素重复, 而后者允许。

- 1) 不能直接改变元素值, 因为那样会打乱原本正确的顺序, 要改变元素值必须先删除旧元素, 则插入新元素
- 2) 不提供直接存取元素的任何操作函数, 只能通过迭代器进行间接存取, 而且从迭代器角度来看, 元素值是常数
- 3) 元素比较动作只能用于型别相同的容器(即元素和排序准则必须相同)

set模板原型: //Key为元素(键值)类型

### 一, set:

#### 1, 构造函数

set () ; 创建空的集合, 这是默认的构造函数

set (T\*first,T\*last); 使用地址区间[first, last]来初始化集合

#### 2, 相关操作

set的各成员函数列表如下:

s.begin(); 返回指向第一个元素的迭代器

s.clear(); 清除所有元素

s.count(); 返回某个值元素的个数

s.empty(); 如果集合为空, 返回true

s.end(); 返回指向最后一个元素的迭代器

s.equal\_range(); 返回集合中与给定值相等的上下限的两个迭代器

s.erase(); 删除集合中的元素

s.find(); 返回一个指向被查找到元素的迭代器

s.get\_allocator(); 返回集合的分配器



s.insert(); 在集合中插入元素

s.lower\_bound(); 返回指向大于（或等于）某值的第一个元素的迭代器

s.key\_comp(); 返回一个用于元素间值比较的函数

s.max\_size(); 返回集合能容纳的元素的最大限值

s.rbegin(); 返回指向集合中最后一个元素的反向迭代器

s.rend(); 返回指向集合中第一个元素的反向迭代器

s.size(); 集合中元素的数目

s.swap(); 交换两个集合变量

s.upper\_bound(); 返回大于某个值元素的迭代器

s.value\_comp(); 返回一个用于比较元素间的值的函数

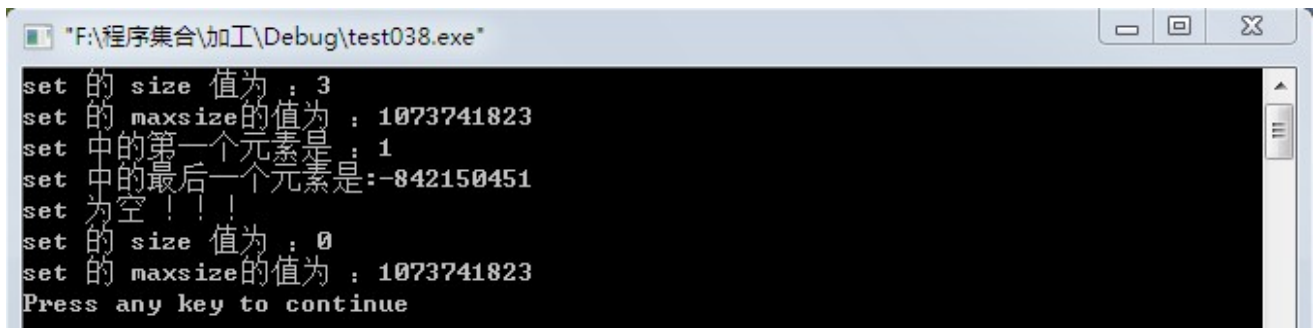
---

应用(转自<http://blog.csdn.net/yas12345678/article/details/52601454>)

#### 一、基本操作

```
1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8     set<int> s;
9     s.insert(1);
10    s.insert(2);
11    s.insert(3);
12    s.insert(1);
13    cout<<"set 的 size 值为 : "<<s.size()<<endl;
14    cout<<"set 的 maxsize的值为 : "<<s.max_size()<<endl;
15    cout<<"set 中的第一个元素是 : "<<*s.begin()<<endl;
16    cout<<"set 中的最后一个元素是:"<<*s.end()<<endl;
17    s.clear();
18    if(s.empty())
19    {
20        cout<<"set 为空 !!! "<<endl;
21    }
22    cout<<"set 的 size 值为 : "<<s.size()<<endl;
23    cout<<"set 的 maxsize的值为 : "<<s.max_size()<<endl;
24    return 0;
25 }
```

运行结果:



```
set 的 size 值为 : 3
set 的 maxsize 的值为 : 1073741823
set 中的第一个元素是 : 1
set 中的最后一个元素是 : -842150451
set 为空 !!!
set 的 size 值为 : 0
set 的 maxsize 的值为 : 1073741823
Press any key to continue
```

**小结：**插入3之后虽然插入了一个1，但是我们发现set中最后一个值仍然是3哈，这就是set。还要注意begin() 和end()函数是不检查set是否为空的，使用前最好使用empty()检验一下set是否为空。

**count()** 用来查找set中某个键值出现的次数。这个函数在set并不是很实用，因为一个键值在set只可能出现0或1次，这样就变成了判断某一键值是否在set出现过了。

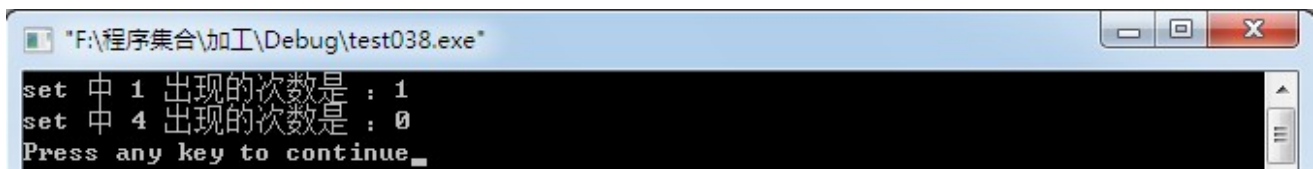
示例代码：



```
1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8     set<int> s;
9     s.insert(1);
10    s.insert(2);
11    s.insert(3);
12    s.insert(1);
13    cout<<"set 中 1 出现的次数是 : "<<s.count(1)<<endl;
14    cout<<"set 中 4 出现的次数是 : "<<s.count(4)<<endl;
15    return 0;
16 }
```



运行结果：



```
set 中 1 出现的次数是 : 1
set 中 4 出现的次数是 : 0
Press any key to continue
```

**equal\_range()**，返回一对定位器，分别表示第一个大于或等于给定关键值的元素和 第一个大于给定关键值的元素，这个返回值是一个pair类型，如果这一对定位器中哪个返回失败，就会等于end()的值。具体这个有什么用途我还没遇到过~~~

示例代码：



```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8     set<int> s;
9     set<int>::iterator iter;
10    for(int i = 1 ; i <= 5; ++i)
11    {
12        s.insert(i);
13    }
14    for(iter = s.begin() ; iter != s.end() ; ++iter)
15    {
16        cout<<*iter<<" ";
17    }
18    cout<<endl;
19    pair<set<int>::const_iterator,set<int>::const_iterator> pr;
20    pr = s.equal_range(3);
21    cout<<"第一个大于等于 3 的数是 : "<<*pr.first<<endl;
22    cout<<"第一个大于 3的数是 : "<<*pr.second<<endl;
23    return 0;
24 }

```



运行结果：

```

F:\程序集合\加工\Debug\test039.exe
1 2 3 4 5
第一个大于等于 3 的数是 : 3
第一个大于 3的数是 : 4
Press any key to continue

```

**erase(iterator)** ,删除定位器iterator指向的值

**erase(first,second)**,删除定位器first和second之间的值

**erase(key\_value)**,删除键值key\_value的值

看看程序吧：



```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8     set<int> s;

```

```

9     set<int>::const_iterator iter;
10    set<int>::iterator first;
11    set<int>::iterator second;
12    for(int i = 1 ; i <= 10 ; ++i)
13    {
14        s.insert(i);
15    }
16    //第一种删除
17    s.erase(s.begin());
18    //第二种删除
19    first = s.begin();
20    second = s.begin();
21    second++;
22    second++;
23    s.erase(first,second);
24    //第三种删除
25    s.erase(8);
26    cout<<"删除后 set 中元素是 : ";
27    for(iter = s.begin() ; iter != s.end() ; ++iter)
28    {
29        cout<<*iter<<" ";
30    }
31    cout<<endl;
32    return 0;
33 }

```



运行结果：

```

"F:\程序集合\加工\Debug\test040.exe"
删除后 set 中元素是 : 4 5 6 7 9 10
Press any key to continue

```

**小结：**set中的删除操作是不进行任何的错误检查的，比如定位器的是否合法等等，所以用的时候自己一定要注意。

**find()**，返回给定值得定位器，如果没找到则返回end()。

示例代码：



```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8     int a[] = {1,2,3};

```

```

9     set<int> s(a,a+3);
10    set<int>::iterator iter;
11    if((iter = s.find(2)) != s.end())
12    {
13        cout<<*iter<<endl;
14    }
15    return 0;
16 }

```



**insert(key\_value);** 将key\_value插入到set中，返回值是pair::iterator,bool>，bool标志着插入是否成功，而iterator代表插入的位置，若key\_value已经在set中，则iterator表示的key\_value在set中的位置。

**inset(first,second);**将定位器first到second之间的元素插入到set中，返回值是void.

示例代码：



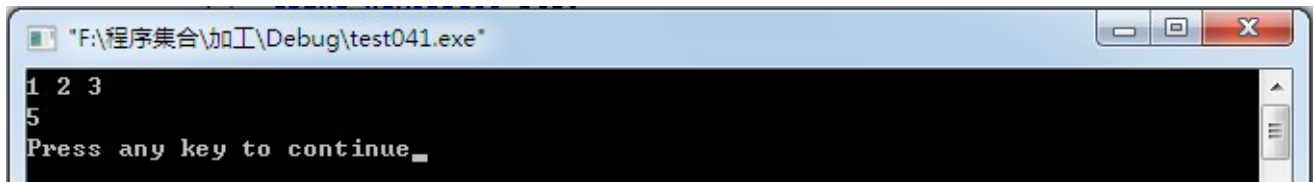
```

1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8     int a[] = {1,2,3};
9     set<int> s;
10    set<int>::iterator iter;
11    s.insert(a,a+3);
12    for(iter = s.begin() ; iter != s.end() ; ++iter)
13    {
14        cout<<*iter<<" ";
15    }
16    cout<<endl;
17    pair<set<int>::iterator,bool> pr;
18    pr = s.insert(5);
19    if(pr.second)
20    {
21        cout<<*pr.first<<endl;
22    }
23    return 0;
24 }

```



运行结果：



```
"F:\程序集合\加工\Debug\test041.exe"
1 2 3
5
Press any key to continue_
```

**lower\_bound(key\_value)** , 返回第一个大于等于key\_value的定位器

**upper\_bound(key\_value)** , 返回最后一个大于等于key\_value的定位器

示例代码:



```
1 #include <iostream>
2 #include <set>
3
4 using namespace std;
5
6 int main()
7 {
8     set<int> s;
9     s.insert(1);
10    s.insert(3);
11    s.insert(4);
12    cout<<*s.lower_bound(2)<<endl;
13    cout<<*s.lower_bound(3)<<endl;
14    cout<<*s.upper_bound(3)<<endl;
15    return 0;
16 }
```



运行结果:



```
"F:\程序集合\加工\Debug\test042.exe"
3
3
4
Press any key to continue
```

## 二.自定义比较函数

(1)元素不是结构体:

例:

//自定义比较函数myComp,重载“<”操作符

[cpp] [view plain copy](#)

```

struct myComp
{
    bool operator()(const your_type &a,const your_type &b)
    {
        return a.data-b.data>0;
    }
}
set<int,myComp>s;
.....
set<int,myComp>::iterator it;

```

(2)如果元素是结构体，可以直接将比较函数写在结构体内。

例：

[cpp] [view plain copy](#)

```

struct Info
{
    string name;
    float score;
    //重载“<”操作符，自定义排序规则
    bool operator < (const Info &a) const
    {
        //按score从大到小排列
        return a.score<score;
    }
}
set<Info> s;
.....
set<Info>::iterator it;

```

## 二，multiset:

### 简介

multiset集合容器：multiset和set的区别：set容器中所有的元素必须独一无二，而multiset容器中元素可以重复，调用头文件：#include，构造函数与set相同在这里不在累述

### 1，相关操作（大部分操作与set相同）

int count( const T & item) const; 返回多重集中匹配的item的个数

pair < iterator,iterator>equal\_range( const T & item); 返回一对迭代器，使得所有匹配的元素都在区间pair的first的成员，pair的second成员之内。

iterator insert (const T & item); 把item添加到多重集中去，返回新元素的迭代器，后置条件：元素item被加入到多重集中去

应用：

```
#include<stdio.h>
#include<set>
using namespace std;
multiset<int> t;
int main(void)
{
    int i;
    for(i=1;i<=3;i++)
        t.insert(2);
    t.insert(1);
    t.insert(4);
    //printf("%d\n", t.size());
    multiset<int>::iterator it;
    for(it=t.begin();it!=t.end();it++)          //输出1 2 2 2 4
        printf("%d ", *it);
    printf("\n");

    it = t.find(2);
    t.erase(t.find(2));
    for(it=t.begin();it!=t.end();it++)          //输出1 2 2 4
        printf("%d ", *it);
    printf("\n");

    printf("%d %d\n", *(t.lower_bound(2)), *(t.upper_bound(2)));    //输出2 4
    return 0;
}
```

来源网址：

<http://blog.csdn.net/yas12345678/article/details/52601454>

<https://www.cnblogs.com/qingtianBKY/p/6663469.html>

## ACM/ICPC 竞赛之 STL--map



## 简介:

在STL的头文件中定义了模板类map和multimap，用有序二叉树来存贮类型为pair的元素对序列。序列中的元素以const Key部分作为标识，map中所有元素的Key值都必须是唯一的，multimap则允许有重复的Key值。可以将map看作是由Key标识元素的元素集合，这类容器也被称为“关联容器”，可以通过一个Key值来快速确定一个元素，因此非常适合于需要按照Key值查找元素的容器。map模板类需要四个模板参数，第一个是键值类型，第二个是元素类型，第三个是比较算子，第四个是分配器类型。其中键值类型和元素类型是必要的。

## 一，map:

### 1, 创建map

创建map需要两个模板参数，这两个模板参数分别对应于pair对象的键分量和值分量。映射的模板参数包括了“键-值”对中键分量的类型名标识符

template

例: map mapobj;

map ID\_Name = { { 2015, "Jim" }, { 2016, "Tom" }, { 2017, "Bob" } };

### 2, 相关操作(转自<http://blog.csdn.net/yas12345678/article/details/52601624>)

1,map添加数据;

map maplive; 1.maplive.insert(pair(102,"active")); 2.maplive.insert(map::value\_type(321,"hai")); 3, maplive[112]="April";//map中最简单最常用的插入添加!

2, map中元素的查找: find()函数返回一个迭代器指向键值为key的元素, 如果没找到就返回指向map尾部的迭代器。 map::iterator l\_it;; l\_it=maplive.find(112); if(l\_it==maplive.end()) cout<<"we do not find 112"<<endl; else cout<<"wo find 112"<<endl; 3,map中元素的删除: 如果删除112; map::iterator l\_it;; l\_it=maplive.find(112); if(l\_it==maplive.end()) cout<<"we do not find 112"<<endl; else maplive.erase(l\_it); //delete 112; 4,map中 swap的用法: Map中的swap不是一个容器中的元素交换, 而是两个容器交换; For example:

```
include <map>

include <iostream>

using namespace std;

int main( )

{

    map <int, int> m1, m2, m3;

    map <int, int>::iterator m1_Iter;

    m1.insert ( pair <int, int> ( 1, 10 ) );

    m1.insert ( pair <int, int> ( 2, 20 ) );
```

```

    m1.insert ( pair <int, int> ( 3, 30 ) );

    m2.insert ( pair <int, int> ( 10, 100 ) );

    m2.insert ( pair <int, int> ( 20, 200 ) );

    m3.insert ( pair <int, int> ( 30, 300 ) );

    cout << "The original map m1 is:";

    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )

        cout << " " << m1_Iter->second;

        cout << "." << endl;

    // This is the member function version of swap

    //m2 is said to be the argument map; m1 the target map

    m1.swap( m2 );

    cout << "After swapping with m2, map m1 is:";

    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )

        cout << " " << m1_Iter -> second;

        cout << "." << endl;

    cout << "After swapping with m2, map m2 is:";

    for ( m1_Iter = m2.begin( ); m1_Iter != m2.end( ); m1_Iter++ )

        cout << " " << m1_Iter -> second;

        cout << "." << endl;

    // This is the specialized template version of swap

    swap( m1, m3 );

    cout << "After swapping with m3, map m1 is:";

    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )

        cout << " " << m1_Iter -> second;

        cout << "." << endl;

}

```

5.map的sort问题: Map中的元素是自动按key升序排序,所以不能对map用sort函数: For example:

```
include <map>

include <iostream>

using namespace std;

int main( )

{

    map <int, int> m1;

    map <int, int>::iterator m1_Iter;

    m1.insert ( pair <int, int> ( 1, 20 ) );

    m1.insert ( pair <int, int> ( 4, 40 ) );

    m1.insert ( pair <int, int> ( 3, 60 ) );

    m1.insert ( pair <int, int> ( 2, 50 ) );

    m1.insert ( pair <int, int> ( 6, 40 ) );

    m1.insert ( pair <int, int> ( 7, 30 ) );

    cout << "The original map m1 is:"<<endl;

    for ( m1_Iter = m1.begin( ); m1_Iter != m1.end( ); m1_Iter++ )

        cout << m1_Iter->first<<" "<<m1_Iter->second<<endl;

}
```

The original map m1 is: 1 20 2 50 3 60 4 40 6 40 7 30 请按任意键继续...

6, map的基本操作函数: C++ Maps是一种关联式容器, 包含“关键字/值”对 begin() 返回指向map头部的迭代器 clear() 删除所有元素 count() 返回指定元素出现的次数 empty() 如果map为空则返回true end() 返回指向map末尾的迭代器 equal\_range() 返回特殊条目的迭代器对 erase() 删除一个元素 find() 查找一个元素 get\_allocator() 返回map的配置器 insert() 插入元素 key\_comp() 返回比较元素key的函数 lower\_bound() 返回键值>=给定元素的第一个位置 max\_size() 返回可以容纳的最大元素个数 rbegin() 返回一个指向map尾部的逆向迭代器 rend() 返回一个指向map头部的逆向迭代器 size() 返回map中元素的个数 swap() 交换两个map upper\_bound() 返回键值>给定元素的第一个位置 value\_comp() 返回比较元素value的函数

## 二, multimap:

multimap大部分操作与map相同, 除了解决相同值得操作, 以下来自

(<http://blog.csdn.net/believefym/article/details/1627874>)

**提出问题** 与 map 不同，multimap 可以包含重复键。这就带来一个问题：重载下标操作符如何返回相同键的多个关联值？以下面的伪码为例：

```
string phone=phonebook["Harry];
```

**标准库**设计者的解决这个问题方法是从 multimap 中去掉下标操作符。因此，需要用不同的方法来插入和获取元素以及和进行错误处理。**插入** 假设你需要开发一个 DNS 后台程序（也就是 Windows 系统中的服务程序），该程序将 IP 地址映射匹配的 URL 串。你知道在某些情况下，相同的 IP 地址要被关联到多个 URLs。这些 URLs 全都指向相同的站点。在这种情况下，你应该使用 multimap，而不是 map。例如：

```
#include <map>
#include <string>
multimap <string, string> DNS_daemon;
```

用 insert() 成员函数而不是**下标操作符**来插入元素。insert()有一个 [pair](#) 类型的参数。在“[使用库创建关联容器](#)”中我们示范了如何使用 make\_pair() 辅助函数来完成此任务。你也可以象下面这样使用它：

```
DNS_daemon.insert(make_pair("213.108.96.7", "cppzone.com"));
```

在上面的 insert()调用中，串“213.108.96.7”是键，“cppzone.com”是其关联的值。以后插入的是相同的键，不同的关联值：

```
DNS_daemon.insert(make_pair("213.108.96.7", "cpluspluszone.com"));
```

因此，DNS\_daemon 包含两个用相同键值的元素。注意 multimap::insert() 和 map::insert() 返回的值是不同的。

```
typedef pair <const Key, T> value_type;
iterator
insert(const value_type&); // #1 multimap
pair <iterator, bool>
insert(const value_type&); // #2 map
```

multimap::insert()成员函数返回指向新插入元素的迭代指针，也就是 iterator（multimap::insert()总是能执行成功）。但是 map::insert() 返回 pair，此处 bool 值表示插入操作是否成功。**查找单个值** 与 map 类似，multimap 具备两个版本重载的 find()成员函数：

```
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
```

find(k) 返回指向第一个与键 k 匹配的 pair 的迭代指针，这就是说，当你想要检查是否存在至少一个与该键关联的值时，或者只需第一个匹配时，这个函数最有用。例如：

```
typedef multimap <string, string> mmss;
void func(const mmss & dns)
{
    mmss::const_iterator cit=dns.find("213.108.96.7");
    if (cit != dns.end())
        cout <<"213.108.96.7 found" <<endl;
    else
        cout <<"not found" <<endl;
}
```

**处理多个关联值**      count(k) 成员函数返回与给定键关联的值得数量。下面的例子报告了有多少个与键“213.108.96.7”关联的值：

```
cout<<dns.count("213.108.96.7") //output: 2
    <<" elements associated"<<endl;
```

为了存取 multimap 中的多个值，使用 equal\_range()、lower\_bound()和 upper\_bound()成员函数：  
equal\_range(k)：该函数查找所有与 k 关联的值。返回迭代指针的 pair，它标记开始和结束范围。下面的例子显示所有与键“213.108.96.7”关联的值：

```
typedef multimap <string, string>::const_iterator CIT;
typedef pair<CIT, CIT> Range;
Range range=dns.equal_range("213.108.96.7");
for(CIT i=range.first; i!=range.second; ++i)
    cout << i->second << endl; //output: cpluspluszone.com
// cppzone.com
```

lower\_bound() 和 upper\_bound()：lower\_bound(k) 查找第一个与键 k 关联的值，而 upper\_bound(k) 是查找第一个键值比 k 大的元素。下面的例子示范用 upper\_bound()来定位第一个其键值大于“213.108.96.7”的元素。通常，当键是一个字符串时，会有一个词典编纂比较：

```
dns.insert(make_pair("219.108.96.70", "pythonzone.com"));
CIT cit=dns.upper_bound("213.108.96.7");
if (cit!=dns.end()) //found anything?
    cout<<cit->second<<endl; //display: pythonzone.com
```

如果你想显示其后所有的值，可以用下面这样的循环：

```
// 插入有相同键的多个值
dns.insert(make_pair("219.108.96.70", "pythonzone.com"));
dns.insert(make_pair("219.108.96.70", "python-zone.com"));
// 获得第一个值的迭代指针
CIT cit=dns.upper_bound("213.108.96.7");
// 输出: pythonzone.com, python-zone.com
while(cit!=dns.end())
{
    cout<<cit->second<<endl;
    ++cit;
}
```

## 结论

虽然 map 和 multimap 具有相同的接口，其重要差别在于重复键，设计和使用要区别对待。此外，还要注意每个容器里 insert()成员函数的细微差别。

---

## 来源网址

<http://blog.csdn.net/chenyujing1234/article/details/8193172>

<http://blog.csdn.net/believefym/article/details/1627874>

# ACM/ICPC 竞赛之 STL--algorithm

---

algorithm意为"算法",是C++的标准模版库 ([STL](#)) 中最重要的头文件之一，内部提供了大量基于迭代器的非成员模板函数，以下内容来自百度百科，不详细介绍，

## 1.不修改内容

<a href="#">adjacent find</a>	查找两个相邻 (Adjacent) 的等价 (Identical) 元素
all_of	检测在给定范围中是否所有元素都满足给定的条件
any_of	检测在给定范围中是否存在元素满足给定条件
count	返回值等价于给定值的元素的个数
<a href="#">count if</a>	返回值满足给定条件的元素的个数
<a href="#">equal</a>	返回两个范围是否相等
<a href="#">find</a>	返回第一个值等价于给定值的元素
find_end	查找范围A中与范围B等价的子范围最后出现的位置
<a href="#">find first of</a>	查找范围A中第一个与范围B中任一元素等价的元素的位置
find_if	返回第一个值满足给定条件的元素
find_if_not	返回第一个值不满足给定条件的元素
for_each	对范围中的每个元素调用指定函数
mismatch	返回两个范围中第一个元素不等价的位置
none_of	检测在给定范围中是否不存在元素满足给定的条件
<a href="#">search</a>	在范围A中查找第一个与范围B等价的子范围的位置
search_n	在给定范围中查找第一个连续 $n$ 个元素都等价于给定值的子范围的位置

## 2.修改内容操作

<a href="#">copy</a>	将一个范围中的元素拷贝到新的位置处
copy_backward	将一个范围中的元素按逆序拷贝到新的位置处
copy_if	将一个范围中满足给定条件的元素拷贝到新的位置处
copy_n	拷贝 n 个元素到新的位置处
<a href="#">fill</a>	将一个范围的元素赋值为给定值
fill_n	将某个位置开始的 n 个元素赋值为给定值
<a href="#">generate</a>	将一个函数的执行结果保存到指定范围的元素中，用于批量赋值范围中的元素
generate_n	将一个函数的执行结果保存到指定位置开始的 n 个元素中
iter_swap	交换两个迭代器（Iterator）指向的元素
move	将一个范围中的元素移动到新的位置处
move_backward	将一个范围中的元素按逆序移动到新的位置处
random_shuffle	随机打乱指定范围中的元素的位置
<a href="#">remove</a>	将一个范围中值等价于给定值的元素删除
remove_if	将一个范围中值满足给定条件的元素删除
remove_copy	拷贝一个范围的元素，将其中值等价于给定值的元素删除
remove_copy_if	拷贝一个范围的元素，将其中值满足给定条件的元素删除
<a href="#">replace</a>	将一个范围中值等价于给定值的元素赋值为新的值
replace_copy	拷贝一个范围的元素，将其中值等价于给定值的元素赋值为新的值
replace_copy_if	拷贝一个范围的元素，将其中值满足给定条件的元素赋值为新的值
replace_if	将一个范围中值满足给定条件的元素赋值为新的值
<a href="#">reverse</a>	反转排序指定范围中的元素
reverse_copy	拷贝指定范围的反转排序结果
<a href="#">rotate</a>	循环移动指定范围中的元素
rotate_copy	拷贝指定范围的循环移动结果
<a href="#">shuffle</a>	用指定的随机数引擎随机打乱指定范围中的元素的位置
swap	交换两个对象的值
<a href="#">swap_ranges</a>	交换两个范围的元素
<a href="#">transform</a>	对指定范围中的每个元素调用某个函数以改变元素的值



<a href="#">copy</a>	将一个范围中的元素拷贝到新的位置处
unique	删除指定范围中的所有连续重复元素，仅仅留下每组等值元素中的第一个元素。
unique_copy	拷贝指定范围的唯一化（参考上述的 unique）结果

### 3.划分操作

<a href="#">is_partitioned</a>	检测某个范围是否按指定谓词（Predicate）划分过
<a href="#">partition</a>	将某个范围划分为两组
partition_copy	拷贝指定范围的划分结果
partition_point	返回被划分范围的划分点
stable_partition	稳定划分，两组元素各维持相对顺序

### 4.排序操作

is_sorted	检测指定范围是否已排序
is_sorted_until	返回最大已排序子范围
<a href="#">nth_element</a>	部分排序指定范围中的元素，使得范围按给定位置处的元素划分
<a href="#">partial_sort</a>	部分排序
partial_sort_copy	拷贝部分排序的结果
<a href="#">sort</a>	排序
stable_sort	稳定排序

### 5.查找操作

binary_search	判断范围中是否存在值等价于给定值的元素
<a href="#">equal_range</a>	返回范围中值等于给定值的元素组成的子范围
<a href="#">lower_bound</a>	返回指向范围中第一个值大于或等于给定值的元素的迭代器
<a href="#">upper_bound</a>	返回指向范围中第一个值大于给定值的元素的迭代器

### 6.集合操作

<b>includes</b>	<b>判断一个集合是否是另一个集合的子集</b>
inplace_merge	就绪合并
<a href="#">merge</a>	合并
<a href="#">set_difference</a>	获得两个集合的差集
set_intersection	获得两个集合的交集
set_symmetric_difference	获得两个集合的对称差
set_union	获得两个集合的并集

## 7.堆操作

<b>is_heap</b>	<b>检测给定范围是否满足堆结构</b>
is_heap_until	检测给定范围中满足堆结构的最大子范围
make_heap	用给定范围构造出一个堆
pop_heap	从一个堆中删除最大的元素
push_heap	向堆中增加一个元素
sort_heap	将满足堆结构的范围排序

## 8.最大最小操作

<b>is_permutation</b>	<b>判断一个序列是否是另一个序列的一种排序</b>
<a href="#">lexicographical_compare</a>	比较两个序列的字典序
max	返回两个元素中值最大的元素
max_element	返回给定范围中值最大的元素
<a href="#">min</a>	返回两个元素中值最小的元素
<a href="#">min_element</a>	返回给定范围中值最小的元素
minmax	返回两个元素中值最大及最小的元素
minmax_element	返回给定范围中值最大及最小的元素
next_permutation	返回给定范围中的元素组成的下一个按字典序的排列
prev_permutation	返回给定范围中的元素组成的上一个按字典序的排列

可查找网址：

<http://www.cplusplus.com/reference/algorithm/>

简单例子：(重点)

<http://blog.csdn.net/tianshuai1111/article/details/7674327#comments>

附：

第10篇 ACM/ICPC竞赛之算法策略 ACM/ICPC竞赛其实就是算法设计和编码的竞赛，熟悉各种常用算法和算法设计策略并能灵活运用是非常必要的。这里对几种在竞赛中经常用到的算法设计策略做一简单的介绍。1、穷举法 穷举法是最基本的算法设计策略，其思想是列举出问题所有的可能解，逐一进行判别，找出满足条件的解。穷举法的运用关键在于解决两个问题：如何列举所有的可能解；如何判别可能解是否满足条件；在运用穷举法时，容易出现的问题是可能解过多，导致算法效率很低，这就需要对列举可能解的方法进行优化。以题1041--纯素数问题为例，从1000到9999都可以看作是可能解，可以通过对所有这些可能解逐一进行判别，找出其中的纯素数，但只要稍作分析，就会发现其实可以大幅度地降低可能解的范围。根据题意易知，个位只可能是3、5、7，再根据题意可知，可以在3、5、7的基础上，先找出所有的二位纯素数，再在二位纯素数基础上找出三位纯素数，最后在三位纯素数的基础上找出所有的四位纯素数。2、分治法 分治法也是应用非常广泛的一种算法设计策略，其思想是将问题分解为若干子问题，从而可以递归地求解各子问题，再综合出问题的解。分治法的运用关键在于解决三个问题：确定分治规则，即如何分解问题。确定终结条件，即问题分解到什么状态时可以直接求解。确定归纳方法，即如何由子问题的解得到原问题的解。这一步并不总是需要的，因为对某些问题来说，并不需要对子问题的解进行复杂的归纳。我们熟知的如汉诺塔问题、折半查找算法、快速排序算法等都是分治法运用的典型案例。以题1045--Square Coins为例，先对题意进行分析，可设一个函数 $f(m, n)$ 等于用面值不超过 $n^2$ 的货币构成总值为 $m$ 的方案数，则容易推导出： $f(m, n) = f(m - 0nn, n - 1) + f(m - 1nn, n - 1) + f(m - 2nn, n - 1) + \dots + f(m - knn, n - 1)$  这里的 $k$ 是币值为 $n^2$ 的货币最多可以用多少枚，即 $k = m / (n * n)$ 。也很容易分析出， $f(m, 1) = f(1, n) = 1$  对于这样的题目，一旦分析出了递推公式，程序就非常好写了。所以在动手开始写程序之前，分析工作做得越彻底，逻辑描述越准确、简洁，写起程序来就会越容易。3、动态规划法 动态规划法多用来计算最优问题，动态规划法与分治法的基本思想是一致的，但处理的手法不同。动态规划法在运用时，要先对问题的分治规律进行分析，找出终结子问题，以及子问题向父问题归纳的规则，而算法则直接从终结子问题开始求解，逐层向上归纳，直到归纳出原问题的解。动态规划法多用于在分治过程中，子问题可能重复出现的情况，在这种情况下，如果按照常规的分治法，自上向下分治求解，则重复出现的子问题就会被重复地求解，从而增大了冗余计算量，降低了求解效率。而采用动态规划法，自底向上求解，每个子问题只计算一次，就可以避免这种重复的求解了。动态规划法还有另外一种实现形式，即备忘录法。备忘录的基本思想是设立一个称为备忘录的容器，记录已经求得解的子问题及其解。仍然采用与分治法相同的自上向下分治求解的策略，只是对每一个分解出的子问题，先在备忘录中查找该子问题，如果备忘录中已经存在该子问题，则不须再求解，可以从备忘录中直接得到解，否则，对子问题递归求解，且每求得一个子问题的解，都将子问题及解存入备忘录中。例如，在题1045--Square Coins中，可以采用分治法求解，也可以采用动态规划法求解，即从 $f(m, 1)$ 和 $f(1, n)$ 出发，逐层向上计算，直到求得 $f(m, n)$ 。在竞赛中，动态规划和备忘录的思想还可以有另一种用法。有些题目中的可能问题数是有限的，而在一次运行中可能需要计算多个测试用例，可以采用备忘录的方法，预先将所有的问题的解记录下来，然后输入一个测试用例，就查备忘录，直接找到答案输出。这在各问题之间存在父子关系的情况下，会更有效。例如，在题1045--Square Coins中，题目中已经指出了最大的目标币值不超过300，也就是说问题数只有300个，而且各问题的计算中存在重叠的子问题，可以采用动态规划法，将所有问题的解先全部计算出来，再依次输入测试用例数据，并直接输出答案。4、回溯法 回溯法是基于问题状态树搜索的求解法，其可适用范围很广。从某种角度上说，可以把回溯法看作是优化了的穷举法。回溯法的基本思想是逐步构造问题的可能解，一边构造，一边用约束条件进行判别，一旦发现已经不可能构造出满足条件的解了，则退回上一步构造过程，重新进行构造。这个退回的过程，就称之为“回溯”。回溯法在运用时，要解决的关键问题在于：如何描述局部解。如何

扩展局部解和回溯局部解。如何判别局部解。回溯法的经典案例也很多，例如全排列问题、N后问题等。5、贪心法 贪心法也是求解最优问题的常用算法策略，利用贪心法策略所设计的算法，通常效率较高，算法简单。贪心法的基本思想是对问题做出目前看来最好的选择，即贪心选择，并使问题转化为规模更小的子问题。如此迭代，直到子问题可以直接求解。基于贪心法的经典算法例如：哈夫曼算法、最小生成树算法、最短路径算法等。但是，贪心法的运用是有条件的，必须能够证明贪心选择能够导出最优解，且转化出的子问题与原问题是同性质的问题，才能使用贪心法求解。一个比较经典的贪心法求解的问题就是找硬币问题：有1、2、5、10、20、50、100七种面值的硬币，要支付指定的金额，问怎么支付所用的硬币个数最少。这是一个非常日常化的问题，凭直觉我们会想到，尽可能先用大面值的硬币，这就是“贪心选择”，而在这个问题上，这个贪心选择也是正确的。6、限界剪枝法 限界剪枝法是求解较复杂最优问题的一种算法策略，与回溯法类似的是，限界剪枝法也是在问题状态空间树上进行搜索，但回溯法是搜索一般解，而限界剪枝法则是搜索最优解。限界剪枝法的基本思想是通过找出权值函数的上下界函数，以下界函数来指导搜索的方向，以上界函数来帮助剪除一些不可能含有最优解的分枝。关于算法和算法策略的讨论是一个非常庞大的话题，几乎每个问题点都能扩展出一大堆可讨论的内容和案例。我实在不知道该怎么用简短的几篇文字就能够把这个话题说透，这里只能蜻蜓点水地对竞赛中经常用到的几种策略做一极为简略的介绍。也许我们可以在以后的文章中，针对具体的题目进行算法和策略的分析，效果可能会更好。

第

11

篇

ACM/ICPC

竞赛之调试

在写程序时，调试程序也是一个重要的环节。怎样才能够更有效地调试程序，发现并修正错误呢？

1

、调试中的输入输出

为了调试程序，我们可能需要反复执行程序，也就需要反复输入相同或不相同的测试数据。如果每次调试运行时都是以手工的方式输入测试数据，相信很多人都会觉得不胜其烦。其实我们可以用一些辅助的手段来简化这个过程。

方法一：使用剪贴板

可以将输入数据预先写好（用记事本、开发环境的编辑器或随便什么能够录入的东西），再将输入数据复制到剪贴板上（也就是说我们通常所说的复制操作）。在调试运行时，就可以直接将输入数据粘贴上去，不需要手工输入，这对于反复调试同一组测试数据尤其方便。

方法二：使用重定向

使用剪贴板对于多组测试数据或者比较长的测试数据就会显得不那么好用了。而使用输入输出的重定向则会更方便。

输入输出重定向是在终端窗口下的一种命令行功能，在命令行上可以用

"<"

表示输入重定向，在

"<"

后跟随输入文件名，则程序将从指定的输入文件中获取输入数据，而不再从键盘读入数据。也可以用

">"

表示输出重定向。在

">"

后跟输出文件名，则程序产生的标准输出将写入指定的输出文件中，而不是显示在屏幕上。

我们可以预先将输入数据存到文本文件中（如果有多组测试数据，可以存成多个文件），用重定向指定准备使用的输入数据。

例如，程序名为

myprog

，输入数据已经存到文件

test.txt

中，则在命令行下可以这样执行：

C:>myprog < test.txt

则程序会直接从

test.txt

中读取输入。如果想把输出结果也存到文件中（这在输出结果比较多时尤其有用，因为直接输出到屏幕上可能会来不及看到输出，或看不全所有的输出），例如，可以这样执行：

C:>myprog > test.out

这样我们就可以在执行后，用一个文本编辑器打开输出文件，慢慢阅读和分析输出结果。

如果把输入和输出的重定向结合起来，也可以这样执行：

C:>myprog < test.txt > test.out 2

### 、输出调试信息

在调试时，很多同学往往首先想到的是使用开发环境所提供的调试功能：设置断点、单步执行、查看和修改变量，甚至改变程序的流程。不可否认，使用开发环境所提供的调试功能的确很方便，但当你过分依赖于这些集成工具时，你可能忽略了很多更有效的手段：仔细地分析、充分的信息。

当我们发现程序没有按照自己预期那样工作时，不要急于跟踪甚至修改程序，而是应该首先仔细对程序的逻辑、语句、表达式进行检查和分析，尽可能使程序在表达上更简洁、更干净。如果实在难以发现问题所在，也不必急于借助于集成工具去跟踪程序的运行。早期的程序员在调试程序时经常会在程序中加入输出调试信息的语句或过程，用以观察程序的运行过程，分析程序的运行逻辑，这种调试手段即使在今天也仍然是非常有效的。

输出的调试信息要尽量容易阅读，格式清楚，在必要的时候，可以借助工具程序或自己编写的程序对输出信息进行处理，以帮助分析问题。

## 3

### 、发现线索

调试的目的就是要分析错误发生的原因，寻找线索。盲目的调试只会浪费时间。

调试中的技巧很多，这里提出几条基本原则：

首先是要使错误可重现，要设法保证能够使错误按照自己的意愿重复出现。对于不知道什么时候会冒出来的错误，分析起来会困难得多！

缩小导致错误的输入，设法构造出最小的又能保证错误出现的输入，这样可以减少变化的可能性，使分析范围更集中。经常可以采用二分选择的方法来选择输入，就是舍掉一半输入，看看错误是否会出现，如果不出现，则选择另一半输入，如此反复，并不断缩小导致错误的输入。

4

#### 、构造测试数据和测试程序

在题目中所给出的测试样例只是一小组测试数据，这虽然通常是我们用来测试程序的第一组数据，但却是远远不够的。我们应该根据题意自行构造更多的测试数据，尤其是一些边界状态的测试数据（数据极大、数据极小、数据量极多、数据量极少、预期出现极端结果等情况）。

边界测试数据可以用于检查程序中是否存在边界错误，设计有缺陷的程序，在处理边界测试数据时往往容易暴露出错误。但如果没有发生明显的运行错误，就需要对结果的正确性进行验证。

有些测试数据可以通过手工计算求出结果，再与程序的计算结果相对比，而也有些问题，可以通过构造测试程序来进行验证。

测试程序通常是用确定可靠的算法编写的解题程序，但不须考虑时间和空间的消耗，用测试程序对测试数据进行求解，用计算结果与待测试程序的计算结果进行对比。

以题

1041--

纯素数问题为例，我们可以用最简单的穷举法进行求解，也许这样的解法是不被接受的，因为效率太低，但这个解法却可以用作我们的测试程序，甚至

——

有同学索性在本地先用这个程序把结果算出来，再写一个程序直接输出结果

——

居然也被接受了！