|  |  |
| --- | --- |
| 标签：  [杂谈](http://search.sina.com.cn/?c=blog&q=%D4%D3%CC%B8&by=tag) | 分类： [编程语言C/C /PERL/Tcl](http://blog.sina.com.cn/s/articlelist_2160998997_3_1.html) |

转自：http://www.cnblogs.com/fangyukuan/archive/2010/09/21/1832364.html

http://www.cplusplus.com/reference/stl/list/

STL: C++ standard template library C++标准模板库

[stl list使用说明](http://blog.sina.com.cn/zhangjian33446/article/details/6085742)

使用标准的std::list进行容器数据处理时，操作比较底层。我们可以，减少引用标准MFC标准库，减少系统的大小，但同时也存在有不方便的操作之处，这里同大家分享一些使用心得......

在使用std::list<>链表时，难免会对数据进行添加删除操作。而遍历链表则有两种方式：通过索引访问，象数组一样处理；通过std::list<>::iterator链表遍历器进行访问

list

STL 中的list 就是一 双向链表，可高效地进行插入删除元素。

list不支持随机访问。所以没有 at(pos)和operator[]。

list 对象list1, list2 分别有元素list1(1,2,3),list2(4,5,6) 。list< int>::iterator it;

|  |  |
| --- | --- |
| list成员 | 说明 |
| constructor | 构造函数 |
| destructor | 析构函数 |
| operator= | 赋值重载运算符 |
| assign | 分配值 |
| front | 返回第一个元素的引用 |
| back | 返回最后一元素的引用 |
| begin | 返回第一个元素的指针(iterator) |
| end | 返回最后一个元素的下一位置的指针 |
| rbegin | 返回链表最后一元素的后向指针(reverse\_iterator or const) |
| rend | 返回链表第一元素的下一位置的后向指针 |
| push\_back | 增加一元素到链表尾 |
| push\_front | 增加一元素到链表头 |
| pop\_back | pop\_back()删除链表尾的一个元素 |
| pop\_front | 删除链表头的一元素 |
| clear | 删除所有元素 |
| erase | 删除一个元素或一个区域的元素(两个重载) |
| remove | 删除链表中匹配值的元素(匹配元素全部删除) |
| remove\_if | 删除条件满足的元素(遍历一次链表)，参数为自定义的回调函数 |
| empty | 判断是否链表为空 |
| max\_size | 返回链表最大可能长度 |
| size | 返回链表中元素个数 |
| resize | 重新定义链表长度(两重载函数) |
| reverse | 反转链表 |
| sort | 对链表排序，默认升序 |
| merge | 合并两个有序链表并使之有序 |
| splice | 对两个链表进行结合(三个重载函数) 结合后第二个链表清空 |
| insert | 在指定位置插入一个或多个元素(三个重载函数) |
| swap | 交换两个链表(两个重载) |
| unique | 删除相邻重复元素 |

1.list 构造函数

list <int > L0 ; // 空链表

list <int > L1 (9); // 建一个含个默认值是的元素的链表

list <int > L2 (5,1); // 建一个含个元素的链表，值都是

list <int > L3 (L2 ); // 建一个L 2 的 copy 链表

list <int > L4 (L0 .begin (), L0 .end ());// 建一个含 L0 一个区域的元素

2. assign() 分配值，有两个重载

L1. assign ( 4,3); // L1(3,3,3,3)

L1. assign( ++list1.beging(), list2.end()); // L 1(2,3)

3 ． operator= 赋值重载运算符

L1 = list1; // L1 (1,2,3)

4. front() 返回第一个元素的引用

int nRet = list1.front() // nRet = 1

5. back() 返回最后一 元素的引用

int nRet = list1.back() // nRet = 3

6. begin() 返回第一个元素的指针(iterator)

it = list1.begin(); // \*it = 1

7. end() 返回最后一个元素的 下一位置 的指针(list 为空时end()=begin())

it = list1.end();

--it; // \*it = 3

8.rbegin() 返回链表最后一 元素的后向指针(reverse\_iterator or const)

list <int >::reverse\_iterator it = list1 .rbegin (); // \*it = 3

9. rend() 返回链表第一元素的 下一位置 的后向指针

list< int>::reverse\_iterator it = list1 .rend(); // \*(--riter) = 1

10.push\_back() 增加一 元素到链表尾

list1.push\_back( 4) // list1(1,2,3, 4 )

11. push\_front() 增加一 元素到链表头

list1.push\_front( 4) // list1( 4 ,1,2,3)

12. pop\_back() 删除链表尾的一个元素

list1.pop\_back( ) // list1(1,2)

13.pop\_front() 删除链表头 的一 元素

list1.pop\_front() 　 // list1(2,3)

14 ．clear() 删除所有元素

list1.clear(); // list1 空了,list1.size() = ０

15.erase() 删除 一个元素 或 一个区域的元素 ( 两个重载函数)

list1.erase( list1.begin()); // list1(2,3)

list1.erase( ++list1.begin(),list1.end()); // list1(1)

16. remove() 删除链表中匹配值 的元素( 匹配元素全部删除)

list 对象L1( 4 ,3,5,1, 4 )

L1.remove( 4); // L1(3,5,1);

17.remove\_if() 删除条件满足的元素( 遍历一次链表) ，参数为自定义的回调函数

// 小于2 的值删除

bool myFun (const int & value ) { return (value < 2); }

list1.remove\_if( myFun ); // list1(3)

18.empty() 判断是否链表为空

bool bRet = L1.empty(); // 若L1 为空，bRet = true ，否则bRet = false 。

19.max\_size() 返回链表最大可能长度

list <int >::size\_type nMax = list1 .max\_size ();// nMax = 1073741823

20 ．size() 返回链表中元素个数

list< int>::size\_type nRet = list1.size(); // nRet = 3

21.resize() 重新定义链表长度( 两重载函数)

list1.resize(5) // list1 (1,2,3, 0,0 ) 用默认值填补

list1.resize(5,4) // list1 (1,2,3, 4,4 ) 用指定值 填补

22.reverse() 反转链表:

list1.reverse( ); // list1(3,2,1)

23.sort() 对链表排序，默认升序( 可自定义回调函数 )

list 对象L1(4,3,5,1,4)

L1.sort( ); // L1(1,3,4,4,5)

L1.sort( greater <int >() ); // L1(5,4,4,3,1)

24.merge() 合并两个有序链表并使之有序

// 升序

list1.merge(list2); // list1(1,2,3,4,5,6) list2 现为空

// 降序

L1( 3,2,1), L2(6,5,4)

L1.merge(L2, greater <int >() ); // list1(6,5,4,3,2,1) list2 现为空

25.splice() 对两个链表进行结合( 三个重载函数) 结合后第二个链表清空

list1.splice( ++list1.begin(),list2);

// list1(1,4,5,6,2,3) list2 为空

list1.splice( ++list1.begin(),list2,list2.begin());

// list1( 1,4,2,3); list2(5,6)

list1.splice( ++list1.begin(),list2,++list2.begin(),list2.end());

//list1( 1, 5,6, 2,3); list2(4)

26.insert() 在指定位置插入一个或多个元素( 三个重载函数)

list1.insert( ++list1.begin(),9); // list1(1,9,2,3)

list1.insert(list1.begin(),2,9); // list1(9,9,1,2,3);

list1.insert(list1.begin(),list2.begin(),--list2.end());//list1(4,5,1,2,3);

27.swap() 交换两个链表( 两个重载)

list1.swap(list2); // list1 （4 ，5 ，6 ） list2 （1 ，2 ，3 ）

28. unique() 删除相邻重复元素

L1( 1, 1 ,4,3,5,1)

L1.unique( ); // L1(1,4,3,5,1)

bool same\_integral\_part (double first , double second )

{ return ( int (first )==int (second ) ); }

L1.unique( same\_integral\_part );

例子：

// -------------------------------------------------------------------------  
// 文件名 : list1.cpp  
// 创建者 : 方煜宽  
// 　邮箱 ： fangyukuan@gmail.com  
// 创建时间 : 2010-9-19 15:58  
// 功能描述 : STL中的list就是一双向链表，可高效地进行插入删除元素。  
//   
// -------------------------------------------------------------------------   
#include " stdafx.h "   
#include < iostream >   
#include < list >   
using namespace std;  
  
list < int > g\_list1;  
list < int > g\_list2;  
  
//////////////////////////////////////////////////////////////////////// //   
  
// 初始化全局链表   
void InitList()  
{  
// push\_back()增加一元素到链表尾   
g\_list1.push\_back( 1 );  
g\_list1.push\_back( 2 );  
g\_list1.push\_back( 3 );  
  
// push\_front()增加一元素到链表头   
g\_list2.push\_front( 6 );  
g\_list2.push\_front( 5 );  
g\_list2.push\_front( 4 );  
}  
  
// 输出一个链表   
void ShowList(list < int >& listTemp)  
{  
// size()返回链表中元素个数   
cout << listTemp.size() << endl;  
  
for (list < int > ::iterator it = listTemp.begin(); it != listTemp.end(); ++ it)  
{  
cout << \* it << ' ' ;  
}  
cout << endl;  
}  
  
//////////////////////////////////////////////////////////////////////// //   
  
// 构造函数，空链表   
void constructor\_test0()  
{  
list < int > listTemp;  
cout << listTemp.size() << endl;  
}  
  
// 构造函数，建一个含三个默认值是0的元素的链表   
void constructor\_test1()  
{  
list < int > listTemp( 3 );  
ShowList(listTemp);  
}  
  
// 构造函数，建一个含五个元素的链表，值都是1   
void constructor\_test2()  
{  
list < int > listTemp( 5 , 1 );  
ShowList(listTemp);  
}  
  
// 构造函数，建一个g\_list1的copy链表   
void constructor\_test3()  
{  
list < int > listTemp(g\_list1);  
ShowList(listTemp);  
}  
  
// 构造函数，listTemp含g\_list1一个区域的元素[\_First, \_Last)   
void constructor\_test4()  
{  
list < int > listTemp(g\_list1.begin(), g\_list1.end());  
ShowList(listTemp);  
}  
  
// assign()分配值，有两个重载  
// template <class InputIterator>  
// void assign ( InputIterator first, InputIterator last );  
// void assign ( size\_type n, const T& u );   
void assign\_test()  
{  
list < int > listTemp( 5 , 1 );  
ShowList(listTemp);  
  
listTemp.assign( 4 , 3 );  
ShowList(listTemp);  
  
listTemp.assign( ++ g\_list1.begin(), g\_list1.end());  
ShowList(listTemp);  
}  
  
// operator=   
void operator\_equality\_test()  
{  
g\_list1 = g\_list2;  
ShowList(g\_list1);  
ShowList(g\_list2);  
}  
  
// front()返回第一个元素的引用   
void front\_test7()  
{  
cout << g\_list1.front() << endl;  
}  
  
// back()返回最后一元素的引用   
void back\_test()  
{  
cout << g\_list1.back() << endl;  
}  
  
// begin()返回第一个元素的指针(iterator)   
void begin\_test()  
{  
list < int > ::iterator it1 = g\_list1.begin();  
cout << \*++ it1 << endl;  
  
list < int > ::const\_iterator it2 = g\_list1.begin();  
it2 ++ ;  
// (\*it2)++; // \*it2 为const 不用修改   
cout << \* it2 << endl;  
  
}  
  
// end()返回 [最后一个元素的下一位置的指针] (list为空时end()= begin())   
void end\_test()  
{  
list < int > ::iterator it = g\_list1.end(); // 注意是：最后一个元素的下一位置的指针   
-- it;  
cout << \* it << endl;  
}  
  
// rbegin()返回链表最后一元素的后向指针   
void rbegin\_test()  
{  
list < int > ::reverse\_iterator it = g\_list1.rbegin();  
for (; it != g\_list1.rend(); ++ it)  
{  
cout << \* it << ' ' ;  
}  
cout << endl;  
}  
  
// rend()返回链表第一元素的下一位置的后向指针   
void rend\_test()  
{  
list < int > ::reverse\_iterator it = g\_list1.rend();  
-- it;  
cout << \* it << endl;  
}  
  
// push\_back()增加一元素到链表尾   
void push\_back\_test()  
{  
ShowList(g\_list1);  
g\_list1.push\_back( 4 );  
ShowList(g\_list1);  
}  
  
// push\_front()增加一元素到链表头   
void push\_front\_test()  
{  
ShowList(g\_list1);  
g\_list1.push\_front( 4 );  
ShowList(g\_list1);  
}  
  
// pop\_back()删除链表尾的一个元素   
void pop\_back\_test()  
{  
ShowList(g\_list1);  
cout << endl;  
  
g\_list1.pop\_back();  
ShowList(g\_list1);  
  
}  
  
// pop\_front()删除链表头的一元素   
void pop\_front\_test()  
{  
ShowList(g\_list1);  
cout << endl;  
  
g\_list1.pop\_front();  
ShowList(g\_list1);  
}  
  
// clear()删除所有元素   
void clear\_test()  
{  
ShowList(g\_list1);  
g\_list1.clear();  
ShowList(g\_list1);  
}  
  
// erase()删除一个元素或一个区域的元素(两个重载函数)   
void erase\_test()  
{  
ShowList(g\_list1);  
g\_list1.erase(g\_list1.begin());  
ShowList(g\_list1);  
  
cout << endl;  
  
ShowList(g\_list2);  
g\_list2.erase( ++ g\_list2.begin(), g\_list2.end());  
ShowList(g\_list2);  
}  
  
// remove()删除链表中匹配值的元素(匹配元素全部删除)   
void remove\_test()  
{  
ShowList(g\_list1);  
g\_list1.push\_back( 1 );  
ShowList(g\_list1);  
  
g\_list1.remove( 1 );  
ShowList(g\_list1);  
}  
  
bool myFun( const int & value) { return (value < 2 ); }  
// remove\_if()删除条件满足的元素(会遍历一次链表)   
void remove\_if\_test()  
{  
ShowList(g\_list1);  
g\_list1.remove\_if(myFun);  
ShowList(g\_list1);  
}  
  
  
// empty()判断是否链表为空   
void empty\_test()  
{  
list < int > listTemp;  
if (listTemp.empty())  
cout << " listTemp为空 " << endl;  
else   
cout << " listTemp不为空 " << endl;  
}  
  
  
// max\_size()返回链表最大可能长度:1073741823   
void max\_size\_test()  
{  
list < int > ::size\_type nMax = g\_list1.max\_size();  
cout << nMax << endl;  
}  
  
  
// resize()重新定义链表长度(两重载函数)：   
void resize\_test()  
{  
ShowList(g\_list1);  
g\_list1.resize( 9 ); // 用默认值填补   
ShowList(g\_list1);  
cout << endl;  
  
ShowList(g\_list2);  
g\_list2.resize( 9 , 51 ); // 用指定值填补   
ShowList(g\_list2);  
}  
  
// reverse()反转链表   
void reverse\_test()  
{  
ShowList(g\_list1);  
g\_list1.reverse();  
ShowList(g\_list1);  
}  
  
  
// sort()对链表排序，默认升序(两个重载函数)   
void sort\_test()  
{  
list < int > listTemp;  
listTemp.push\_back( 9 );  
listTemp.push\_back( 3 );  
listTemp.push\_back( 5 );  
listTemp.push\_back( 1 );  
listTemp.push\_back( 4 );  
listTemp.push\_back( 3 );  
  
ShowList(listTemp);  
listTemp.sort();  
ShowList(listTemp);  
  
listTemp.sort(greater < int > ());  
ShowList(listTemp);  
}  
  
// merge()合并两个升序序链表并使之成为另一个升序.   
void merge\_test1()  
{  
list < int > listTemp2;  
listTemp2.push\_back( 3 );  
listTemp2.push\_back( 4 );  
  
list < int > listTemp3;  
listTemp3.push\_back( 9 );  
listTemp3.push\_back( 10 );  
  
ShowList(listTemp2);  
cout << endl;  
ShowList(listTemp3);  
cout << endl;  
  
listTemp2.merge(listTemp3);  
ShowList(listTemp2);  
}  
  
  
bool myCmp ( int first, int second)  
{ return ( int (first) > int (second) ); }  
  
// merge()合并两个降序链表并使之成为另一个降序.   
void merge\_test2()  
{  
list < int > listTemp2;  
listTemp2.push\_back( 4 );  
listTemp2.push\_back( 3 );  
  
list < int > listTemp3;  
listTemp3.push\_back( 10 );  
listTemp3.push\_back( 9 );  
  
ShowList(listTemp2);  
cout << endl;  
ShowList(listTemp3);  
cout << endl;  
  
// listTemp2.merge(listTemp3, greater<int>()); // 第二个参数可以是自己定义的函数如下   
listTemp2.merge(listTemp3, myCmp);  
ShowList(listTemp2);  
}  
  
// splice()对两个链表进行结合(三个重载函数),结合后第二个链表清空  
// void splice ( iterator position, list<T,Allocator>& x );  
// void splice ( iterator position, list<T,Allocator>& x, iterator i );  
// void splice ( iterator position, list<T,Allocator>& x, iterator first, iterator last );   
void splice\_test()  
{  
list < int > listTemp1(g\_list1);  
list < int > listTemp2(g\_list2);  
  
ShowList(listTemp1);  
ShowList(listTemp2);  
cout << endl;  
  
//   
listTemp1.splice( ++ listTemp1.begin(), listTemp2);  
ShowList(listTemp1);  
ShowList(listTemp2);  
  
//   
listTemp1.assign(g\_list1.begin(), g\_list1.end());  
listTemp2.assign(g\_list2.begin(), g\_list2.end());  
listTemp1.splice( ++ listTemp1.begin(), listTemp2, ++ listTemp2.begin());  
ShowList(listTemp1);  
ShowList(listTemp2);  
  
//   
listTemp1.assign(g\_list1.begin(), g\_list1.end());  
listTemp2.assign(g\_list2.begin(), g\_list2.end());  
listTemp1.splice( ++ listTemp1.begin(), listTemp2, ++ listTemp2.begin(), listTemp2.end());  
ShowList(listTemp1);  
ShowList(listTemp2);  
  
}  
  
// insert()在指定位置插入一个或多个元素(三个重载函数)  
// iterator insert ( iterator position, const T& x );  
// void insert ( iterator position, size\_type n, const T& x );  
// template <class InputIterator>  
// void insert ( iterator position, InputIterator first, InputIterator last );   
void insert\_test()  
{  
list < int > listTemp1(g\_list1);  
ShowList(listTemp1);  
listTemp1.insert(listTemp1.begin(), 51 );  
ShowList(listTemp1);  
cout << endl;  
  
list < int > listTemp2(g\_list1);  
ShowList(listTemp2);  
listTemp2.insert(listTemp2.begin(), 9 , 51 );  
ShowList(listTemp2);  
cout << endl;  
  
list < int > listTemp3(g\_list1);  
ShowList(listTemp3);  
listTemp3.insert(listTemp3.begin(), g\_list2.begin(), g\_list2.end());  
ShowList(listTemp3);  
  
}  
  
// swap()交换两个链表(两个重载)   
void swap\_test()  
{  
ShowList(g\_list1);  
ShowList(g\_list2);  
cout << endl;  
  
g\_list1.swap(g\_list2);  
ShowList(g\_list1);  
ShowList(g\_list2);  
}  
  
bool same\_integral\_part ( double first, double second)  
{ return ( int (first) == int (second) ); }  
  
// unique()删除相邻重复元素   
void unique\_test()  
{  
list < int > listTemp;  
listTemp.push\_back( 1 );  
listTemp.push\_back( 1 );  
listTemp.push\_back( 4 );  
listTemp.push\_back( 3 );  
listTemp.push\_back( 5 );  
listTemp.push\_back( 1 );  
list < int > listTemp2(listTemp);  
  
ShowList(listTemp);  
listTemp.unique(); // 不会删除不相邻的相同元素   
ShowList(listTemp);  
cout << endl;  
  
listTemp.sort();  
ShowList(listTemp);  
listTemp.unique();  
ShowList(listTemp);  
cout << endl;  
  
listTemp2.sort();  
ShowList(listTemp2);  
listTemp2.unique(same\_integral\_part);  
ShowList(listTemp2);  
  
}  
  
// 主函数，下面要测试哪个就把那个注释去掉即可   
int \_tmain( int argc, \_TCHAR \* argv[])  
{  
InitList();  
// ShowList(g\_list1);  
// ShowList(g\_list2);  
  
// constructor\_test0();  
// constructor\_test1();  
// constructor\_test2();  
// constructor\_test3();  
// constructor\_test4();  
// assign\_test();  
// operator\_equality\_test();  
// front\_test7();  
// back\_test();  
// begin\_test();  
// end\_test();  
// rbegin\_test();  
// rend\_test();  
// push\_back\_test();  
// push\_front\_test();  
// pop\_back\_test();  
// pop\_front\_test();  
// clear\_test();  
// erase\_test();  
// remove\_test();  
// remove\_if\_test();  
// empty\_test();  
// max\_size\_test();  
// resize\_test();  
// reverse\_test();  
// sort\_test();  
// merge\_test1();  
// merge\_test2();  
// splice\_test();  
// insert\_test();  
// swap\_test();  
// unique\_test();   
return 0 ;  
}