数据结构与算法

STL初步

王　昭
北京大学计算机学院

wangzhao@pku.edu.cn

STL

- STL（Standard Template Library）是标准模板库的缩写。在STL中，使用类模板实现的变长数组、链表、栈等数据结构，被称为容器。
- STL中的容器分为3大类：顺序容器、关联容器和容器适配器。
- STL中也包括一些常用的算法，如排序、查找、交换等。

容器用法

- vector<int>是一个容器类的名字
- vector<int> a; 定义了一个容器对象
- 任何两个容器对象，只要类型相同，就可以用<,<=,>,>=,==,!=进行词典式的比较运算

顺序容器

- 元素在容器中的值与位置无关，即容器是不排序的。
  - 包括vector、deque和list等3种容器。
  - vector和deque都表示变长的数组结构。
  - list表示双向链表结构。

顺序容器的成员函数-1

- **int size()**:返回容器对象中元素的个数
- **bool empty()**: 判断容器对象是否为空

所有容器都有的成员函数

- **front()**:返回容器中第一个元素的引用
- **back()**:返回容器中最后一个元素的引用
- **push_back()**:在容器末尾增加新元素
- **pop_back()**:在容器末尾删除新元素
- **void clear()**:删除所有元素

顺序容器的iterator

- 访问容器中的数据元素要通过容器的“迭代器”
- “*迭代器名”表示迭代器指向的元素
- 容器list支持双向迭代器。即可向后、向前移动1个位置。
- 容器vector和deque支持随机访问迭代器，即在不超出容器范围的前提下，可以向后、向前移动任意个位置。

迭代器的定义

- 正向迭代器　容器类名::iterator 迭代器名;
- 常量正向迭代器　容器类名::const_iterator 迭代器名;
- 反向迭代器　容器类名::reverse_iterator 迭代器名;
- 常量反向迭代器　容器类名::const_ reverse_iterator 迭代器名;

迭代器支持的运算

运算（p、p1表示迭代器，i表示整数）	说明
++p、p++、--p、p--	自增1、自减1
p+=i、p-=i	自增i、自减i，仅随机访问迭代器支持
p+i、p-i	加i、减i，仅随机访问迭代器支持
p[i]	*(p+i)，仅随机访问迭代器支持
*p	引用迭代器的元素
p=p1	赋值运算
p==p1、p!=p1	比较是否相等
p<p1、p<=p1、p>p1、p>=p1	比较大小，仅随机访问迭代器支持

顺序容器的成员函数-2

- begin()：返回指向容器中第一个元素的迭代器
- end()：返回指向容器中最后一个元素后面的位置的迭代器
- rbegin():返回指向容器中最后一个元素的反向迭代器
- rend():返回指向容器中第一个元素前面的位置的迭代器
- erase():从容器中删除一个或几个元素
- insert():插入一个或多个元素

顺序容器-vector

- vector表示一个可变长数组。在预分配的存储空间存满以后，再插入新数据时，存储空间自动翻倍。
- vector 将所有数据元素保存在一块连续内存中。因此，它的优点是支持快速的随机访问。
- 在中间位置插入或删除元素的速度很慢。
- push_back()和pop_back()都很高效。

```
1.vector<类型>标识符
2.vector<类型>标识符(最大容量)
3.vector<类型>标识符(最大容量,初始所有值)
```


```cpp
#include <stdio.h>
#include <vector>

using namespace std;

int main()
{
    vector<int> va;
    for(int n=0;n<6;n++)
        va.push_back(n);

    vector<int>::iterator i;
    for(i=va.begin();i!=va.end();i++){
        printf("%d ",*i);
        *i+=3;
    }
    printf("\n");

    vector<int>::reverse_iterator j;
    for(j=va.rbegin();j!=va.rend();j++){
        printf("%d ",*j);
    }
    return 0;
}
```

1.vector<类型> 标识符

```
0 1 2 3 4 5
8 7 6 5 4 3
```


1.vector<类型>标识符(最大容量)

```
#include <stdio.h>
#include <vector>

using namespace std;

int main()
{
    vector<int> va(20);
    for(int n=0;n<va.size();n++){
        printf("%d ",va[n]);
    }
    printf("\n");

    vector<int>::iterator i;
    for(i=va.begin();i!=va.end();i++){
        printf("%d ",*i);
    }
    printf("\n");

    for(i=va.begin();i<va.end();i++){
        printf("%d ",*i);
    }
    return 0;
}
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

vector内存增长机制

- 可以自动地动态调整大小以适应存储的元素数量变化

```
vector<int> a;
// size() returns the number of elements in the vector
// capacity() returns the size of the storage space currently allocated for the vector,
// expressed in terms of elements
cout << a.size() << " " << a.capacity() << endl; // 1
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 2
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 3
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 4
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 5
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 6
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 7
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 8
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 9
a.push_back(0);
cout << a.size() << " " << a.capacity() << endl; // 10
```

输出

```
0 0
1 1
2 2
3 4
4 4
5 8
6 8
7 8
8 8
9 16
```


关于vector的clear()函数的一个细节

```
struct Student
{
    /* data */
    int id;
    int score;

    Student(int _id, int _score){
        id = _id;
        score = _score;
    }
};

int main(int argc, char **argv){
    vector<Student> stu_s;
    stu_s.push_back(Student(0, 100));
    stu_s.push_back(Student(1, 99));
    stu_s.push_back(Student(2, 80));
    stu_s.push_back(Student(3, 100));

    cout<<"before clear\n";
    cout << stu_s.size() << endl;
    cout << stu_s.capacity() << endl;

    // call clear
    stu_s.clear();

    cout<<"after clear\n";
    cout << stu_s.size() << endl;
    cout << stu_s.capacity() << endl;
    return 0;
}
```

输出

```
before clear
4
4
after clear
0
4
```


vector成员函数	作用
vector()	容器初始化为空
vector(int n)	容器初始化为n个元素
vector(iterator first, iterator end)	容器初始化为与其他容器的区间[first,end]一致
vector(int n，const T & Val)	容器初始化为n个元素，假定元素类型是T，元素的值为Val
iterator insert(iterator i,const T & Val)	Val插入位置i
iterator insert(iterator i, iterator first, iterator end)	其他容器的区间[first,end]中的元素插入位置i
iterator erase(iterator i)	删除位置i的元素
iterator erase(iterator first, iterator end)	删除区间[first,end]中的元素

```cpp
#include <stdio.h>
#include <vector>
using namespace std;

int main()
{
    int aa[5]={1,2,3,4,5};
    vector<int> va(aa,aa+5);//数组aa的内容放入va
    va.insert(va.begin()+2,16);
    vector<int>::iterator i;
    for(i=va.begin();i!=va.end();i++)
        printf("%d ",*i);
    printf("\n");
    va.erase(va.begin()+2);
    for(i=va.begin();i!=va.end();i++)
        printf("%d ",*i);
    printf("\n");
```

```
1 2 16 3 4 5
1 2 3 4 5
```


```
    vector<int> vb(4,100);    //vb有四个元素都是100
    vb.insert(vb.begin(),va.begin()+1,va.begin()+3);
    for(i=vb.begin();i!=vb.end();i++)
        printf("%d ",*i);
    printf("\n");
    va.erase(va.begin()+1,va.begin()+3);
    for(i=va.begin();i!=va.end();i++)
        printf("%d ",*i);
    return 0;
}
```

```
2  3  100  100  100  100
1  4  5
```


顺序容器-list

- list表示一个链表的结构，它的底层存储结构采用的是双向链表。
- list不支持随机访问，但是与vector和deque相比，list容器在任意位置插入或删除元素都是高速的。

list成员函数	作用
void push_front(const T & Val)	Val插在链表最前面
void pop_front( )	删除链表最前面的元素
void remove(const T & Val)	删除和Val相等的元素
void unique( )	删除所有和前一个元素相等的元素
void merge(list <T> &x)	将链表x合并进来，并清空x，要求链表有序

迭代器的辅助函数

- 需要包含头文件# include <algorithm>

advance(p,n)：使迭代器p向前或向后移动n个元素

distance(p,q)：计算两个迭代器之间的距离

iter_swap(p,q)：用于交换两个迭代器p、q指向的值

```
# include <list>
# include <iostream>
# include <algorithm>

using namespace std;

int main()
{
    int a[5]={1,2,3,4,5};
    list<int> lst(a,a+5);
    list<int> ::iterator p=lst.begin();
    advance(p,2);    //后移2个元素，指向3
    printf("1) %d\n",*p);
    advance(p,-1);       //前移1个元素，指向2
    printf("2) %d\n",*p);
    list<int> ::iterator q=lst.end();
    q--;      //指向5
    printf("3) %d\n",*p);
    iter_swap(p,q);     //交互2和5
    printf("4) ",*p);
    for(p=lst.begin();p!=lst.end();p++)
       printf("%d ",*p);
    return 0;
}
```

```
1) 3
2) 2
3) 2
4) 1 5 3 4 2
```


顺序容器-deque

- deque也表示一个可变长的数组。
- deque将数据元素保存在多块连续的内存中，同时在一个映射结构中保存对这些内存块的跟踪，以及它们的先后顺序。
- 在头尾都能较快增删元素
- `void push_front(const T& val); //将val插入容器头部`
- `void pop_front();  //删除容器头部元素`

容器适配器

- STL中的容器适配器stack、queue、 priority_queue
- push(): 添加一个元素
- top():返回顶部（对stack而言）或队头（对queue而言）的元素
- pop:删除一个元素
- size():返回容器适配器元素个数
- empty():判空

容器适配器stack

```
#include <stdio.h>
#include <stack>  //使用stack需要包含此头文件

using namespace std;

void conversion(int m );

int main()
{
    int m;
    scanf ("%d",&m);
    conversion(m);
    return 0;

}
```


```
void conversion(int m )
{
        int n,e;
        stack<int>stk;
        printf("please input the number to be converted n=:");
        scanf ("%d",&n);
        while(n){
             stk.push(n%m);
             n=n/m;
        }
        while(!stk.empty()){
             e=stk.top();
             stk.pop();
             printf("%d",e);
        }
}
```


容器适配器queue

```
#include <stdio.h>
#include <queue>
using namespace std;

int main()
{
    queue<int> q1;
    for(int i=1;i<=5;++i)
    q1.push(i);
    printf("%d\n",q1.size());
    while(!q1.empty())
    {   printf("%d ",q1.front());
        q1.pop();
    }
    printf("\n%d\n",q1.size());
    return 0;
}
```

容器适配器priority_queue

- 容器适配器priority_queue表示大根堆，最“大”元素总是最先被删除，这里的“大”是指在某种比较大小方式下的“大”。
- 用户可以通过显式提供模板参数改变priority_queue默认的比较大小方式。

```
priority_queue <double,vector<double>,less<double> >
priority_queue <double,deque<double>,greater<double> >
```

- 容器适配器priority_queue可以使用顺序容器vector和deque适配，默认使用vector。
- 插入和删除元素的复杂度都是O(logn)

```cpp
#include <queue>
#include <iostream>
using namespace std;

int main(){
    priority_queue <double> pq1;
    pq1.push(3.2);pq1.push(9.6);pq1.push(9.8);pq1.push(5.1);
    while(!pq1.empty()){
        printf("%.1lf ",pq1.top());
        pq1.pop();
    }
    printf("\n");
    priority_queue <double,deque<double>,greater<double> > pq2;
    pq2.push(3.2);pq2.push(9.6);pq2.push(9.8);pq2.push(5.1);
    while(!pq2.empty()){
        printf("%.1lf ",pq2.top());
        pq2.pop();
    }
    return 0;
}
```

```
9.8 9.6 5.1 3.2
3.2 5.1 9.6 9.8
```

关联容器-map

- map提供一对一的key-value映射
- 声明方式：map<key_type, mapped_type>
- find(const key_type& k)：根据key在map中搜寻对应元素，如果找到则返回对应迭代器，否则返回map::end()
- erase():可以根据key或者迭代器删除元素

关联容器-map

```
std::map<char,int> mymap;
std::map<char,int>::iterator it;

mymap['a']=50;
mymap['b']=100;
mymap['c']=150;
mymap['d']=200;

it = mymap.find('b');
if (it != mymap.end())
    mymap.erase (it);

// print content:
std::cout << "elements in mymap:" << '\n';
std::cout << "a => " << mymap.find('a')->second << '\n';
std::cout << "c => " << mymap.find('c')->second << '\n';
std::cout << "d => " << mymap.find('d')->second << '\n';
```

输出

```
elements in mymap:
a => 50
c => 150
d => 200
```


两数之和

给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

思路：使用map存储数组元素与对应下标，再遍历查找target-nums[i]是否在map中

https://leetcode-cn.com/problems/two-sum/

algorithm库

- 包含头文件#include <algorithm>即可使用
- 其中提供了大量基于迭代器的非成员模板函数
- sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp):对[first, last)中的元素使用comp进行排序
- set_union:对有序集合求并
- ……

sort

```
bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;
```

输出

```
myvector contains: 12 26 32 33 45 53 71 80
```

```
int myints[] = {32,71,12,45,26,80,53,33};
std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33

// using default comparison (operator <):
std::sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26 80 53 33

// using function as comp
std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

// using object as comp
std::sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45 53 71 80)

// print out content:
std::cout << "myvector contains:";
for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
```


set_union

```
int first[] = {5,10,15,20,25};
int second[] = {50,40,30,20,10};
std::vector<int> v(10);                      // 0  0  0  0  0  0  0  0  0  0
std::vector<int>::iterator it;

std::sort (first,first+5);     //  5 10 15 20 25
std::sort (second,second+5);   // 10 20 30 40 50

it=std::set_union (first, first+5, second, second+5, v.begin());
                                             // 5 10 15 20 25 30 40 50  0  0
v.resize(it-v.begin());                      // 5 10 15 20 25 30 40 50

std::cout << "The union has " << (v.size()) << " elements:\n";
for (it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
```

输出

```
The union has 8 elements:
 5 10 15 20 25 30 40 50
```


algorithm后续

- find_if:条件查找
- set_union:有序元素序列的求并
- binary_search:有序元素序列的二分查找
- transform:对一定范围内的元素统一实施某种操作
- ……

- 完整algorithm算法列表可以参考
  http://www.cplusplus.com/reference/algorithm/