

标准模版库

(Standard Template Library , STL)

西安交通大学 仇国巍

1、STL概述

- STL是一个具有工业强度的，高效的C++程序库
- 它实现了诸多在计算机科学领域里常用的基本数据结构和基本算法
- STL主要包含了容器、算法、迭代器
- STL系由Alexander Stepanov和Meng Lee等人创造于惠普实验室
- STL于1994年2月年正式成为ANSI/ISO C++的一部份

容器

- ▶ 是容纳、包含相同类型元素的对象，主要用类模板实现
- ▶ 序列型容器：容器中的元素按线性结构组织起来，可以逐个读写元素。主要代表有vector（向量）、deque（双端队列）、list（双向链表）
- ▶ 关联型容器：关联容器通过键（key）存储和读取元素。主要有map（映射）、set（集合）等
- ▶ 容器适配器：是对前面提到的某些容器（如vector）进行再包装，使其变为另一种容器。典型的有栈（stack）、队列（queue）等

迭代器

- ▶ 是用于确定元素位置的数据类型，可用来遍历容器中的元素
- ▶ 通过迭代器可以读取、修改它指向的元素，它的用法和指针类似
- ▶ 每一种容器都定义了一种迭代器

定义一个容器类的迭代器的方法可以是：

容器类名<元素类型>::iterator 变量名;

例如：vector<int>::iterator it;

访问一个迭代器指向的元素：

* 迭代器变量名

例如：*it=5；

算法

- ▶ 由许多函数模版组成的集合，实现了大量通用算法，用于操控各种容器
- ▶ STL中提供的算法涉及到：比较、交换、查找、遍历、复制、修改、移除、反转、排序、合并等。大约有70种标准算法
- ▶ 算法通过迭代器来操纵容器中的元素
- ▶ 算法可以处理容器，也可以处理C语言的数组

2、从了解 vector 开始

vector主要特征

- ▶ vector 实际上就是对动态数组封装
- ▶ 可以像数组一样可以使用下标访问元素，若vector长度为 n ，则其下标为 $0 \sim n-1$
- ▶ 根据下标访问元素效率高
- ▶ vector对象的空间随着插入删除操作自动调整
- ▶ 因为空间自动调整比较耗费时间，因此频繁插入删除的情况下，vector效率稍差

vector—对象创建

- 创建一个空向量

```
vector<int> v1;           // int 类型向量  
vector<string> s1;       // string 类型向量
```

- 从已有向量复制创建向量

```
vector<int> v2( v1 );    // 拷贝v1内容到v2 ( 拷贝构造函数 )
```

- 创建10个元素的向量

```
vector<string> s2( 10 );
```

- 创建10个元素的向量，所有元素都是 1.5

```
vector<double> v3( 10, 1.5 );
```

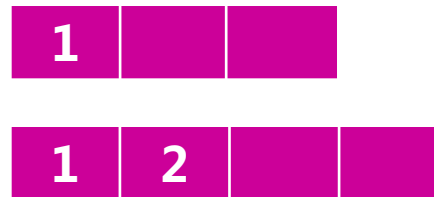
- 创建向量指针

```
vector<int> *pvec = new vector<int>( 10, -5 );
```

vector—尾部添加元素

- 使用 `push_back()` 函数向 `vector` 尾部添加元素

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(2);
    return 0;
}
```



vector—任意位置插入元素

- 使用 insert() 函数向 vector 添加元素

```
#include<iostream>
#include<vector>    // 使用vector必备
using namespace std;
int main()
{
    vector<int> v1;
    v1.push_back(1);  v1.push_back(2);
    v1.insert(v1.begin(),0); //头部插入
    v1.insert(v1.end(), 4);   //尾部插入
    v1.insert(v1.end()-1,3); //倒数第二位置
    return 0;
}
```

1		
---	--	--

1	2		
---	---	--	--

0	1	2		
---	---	---	--	--

0	1	2	4	
---	---	---	---	--

0	1	2	3	4	
---	---	---	---	---	--

v1.begin(), v1.end() 获取相应位置的迭代器

vector—用下标访问元素

- 使用 [下标] 可以获取元素值，修改元素

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> v1;
    .....
    v1.insert(v1.end()-1,3); //倒数第二位置
    v1[4]=10;                // v1[5]=6; 超界错误
    for(int i=0; i<v1.size(); i++)
        cout<<v1[i]<<" ";
    return 0;
}
```

1		
---	--	--

1	2		
---	---	--	--

0	1	2		
---	---	---	--	--

0	1	2	4	
---	---	---	---	--

0	1	2	3	10	
---	---	---	---	----	--

v1.size() 为 5

```
0 1 2 3 10 请按任意键继续.
```

vector—删除尾部元素

- 使用 pop_back() 删除最后一个元素

```
#include<iostream>
```

```
#include<vector>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    vector<int> v1;
```

```
    .....    .....
```

```
    v1[4]=10;
```

```
    v1.pop_back(); //删除 10
```

```
    return 0;
```

```
}
```

1		
---	--	--

1	2		
---	---	--	--

0	1	2		
---	---	---	--	--

0	1	2	4	
---	---	---	---	--

0	1	2	3	10	
---	---	---	---	----	--

0	1	2	3		
---	---	---	---	--	--

vector—删除任意元素

- 使用 erase() 任意位置元素

```
#include<iostream>
```

```
#include<vector>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    vector<int> v1;
```

```
    .....    .....
```

```
    v1.pop_back(); //删除 10
```

```
    v1.erase(v1.begin()); //删除 0
```

```
    v1.erase(v1.begin(), v1.end()); //全删
```

```
    return 0;
```

```
}
```

```
v1.clear(); //全删除
```

1		
---	--	--

1	2		
---	---	--	--

0	1	2		
---	---	---	--	--

0	1	2	4	
---	---	---	---	--

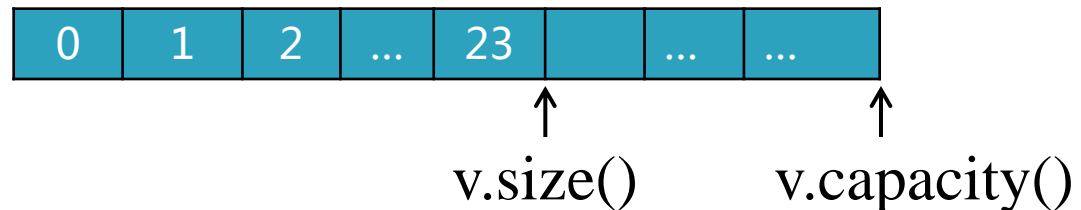
0	1	2	3	10	
---	---	---	---	----	--

1	2	3			
---	---	---	--	--	--

--	--	--	--	--	--

vector—向量大小相关函数

<code>v.size()</code>	返回向量的大小
<code>v.max_size()</code>	返回向量可容纳的最大个数
<code>v.empty()</code>	返回向量是否为空
<code>v.resize(n)</code>	调整向量大小，使其可以容纳n个元素，如果 $n < v.size()$ ，则删除多出来的元素；否则，添加新元素
<code>v.resize(n,t)</code>	调整向量的大小，使其可以容纳n个元素，所有新添加的元素初始化为t
<code>v.capacity()</code>	获取向量的容量，再分配内存空间之前所能容纳的元素个数



3、vector 上的迭代器

迭代器基本操作

- 向量上的迭代器定义、使用

```
vector<int>::iterator it ;    *it = 5;
```

- vector上迭代器支持随机访问：

- 1. 提供读写操作
- 2. 并能在数据中随机移动(前后，跳跃式)

- 用加、减操作移动迭代器：

```
it++;    ++it;           //指向下一元素
it--;    --it;           //指向前一元素
it + i : 返回指向 it 后面的第i个元素的迭代器
it - i : 返回指向 it 前面的第i个元素的迭代器
```

- 用 <, <=, >, >=, ==, != 判断迭代器前后、相等关系：

```
it1 < it2    // 表示 it1 在 it2 之前
```

begin()和end()函数

- 每种容器都定义了一对命名为begin和end的函数，用于返回迭代器。如果容器中有元素，由begin返回的迭代器指向第一个元素：

```
it = v1.begin(); // 指向v1[0]
```

- 由 end 返回的迭代器指向vector的末端元素的下一个。通常称为超出末端迭代器，表明它指向了一个不存在的元素

```
it = v1.end(); // 指向末端元素的下一个
```

- 如果vector为空，begin返回的迭代器与end返回的迭代器相同

用迭代器读取元素

- 将1~9加入vector，再将偶数取出

```
#include<iostream>
```

```
#include<vector>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    vector<int> v1;
```

```
    for(int i=1; i<10; i++)
```

```
        v1.push_back(i); // 添加1~9
```

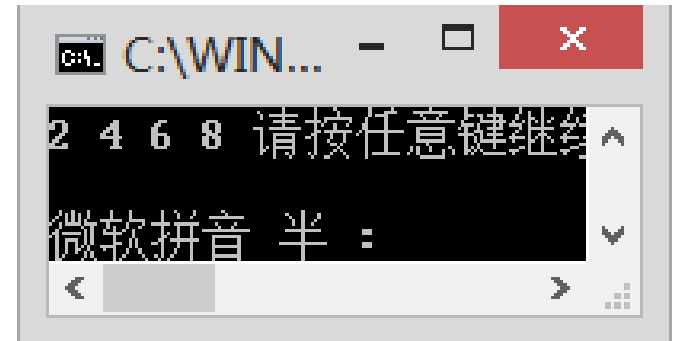
```
    vector<int>::iterator it;
```

```
    for(it=v1.begin(); it<v1.end(); it++)
```

```
        if(*it%2==0) cout<<*it<<" ";
```

```
    return 0;
```

```
}
```



以迭代器为参数的插入删除函数

v.insert(p,t)

在迭代器p所指向的元素前面插入值为 t 的元素

v.insert(p,n,t)

在迭代器p所指向的元素前面插入n个值为t的新元素

v.insert(p,b,e)

在迭代器p所指向的元素前面插入迭代器b和e标记的范围内的元素

v.erase(p)

删除迭代器p指向的容器中的元素

v.erase(b,e)

删除迭代器b和e所标记范围内的元素

通过迭代器进行删除和插入

```
vector<int> v2(3, 1);  
vector<int> v1(4, 0);  
v1.insert(v1.begin(), 5);    // 在头部插入 5  
v1.insert(v1.end(), 7 );    // 在尾部插入 7  
// 在下标为4处插入9  
vector<int>::iterator it = v1.begin() + 4;  
v1.insert(it, 9 );  
// 删除偶数元素  
for(it=v1.begin(); it<v1.end(); ) {  
    if(*it%2==0) it=v1.erase(it);  
    else it++;  
}  
// 将v1的一部分拷贝到v2  
v2.insert(v2.begin(),v1.begin(),v1.begin()+2);
```

1	1	1						v2
0	0	0	0					v1
5	0	0	0	0				v1
5	0	0	0	0	7			v1

5	0	0	0	9	0	7		v1
---	---	---	---	---	---	---	--	----

5	9	7						v1
---	---	---	--	--	--	--	--	----

5	9	1	1	1				v2
---	---	---	---	---	--	--	--	----

用迭代器循环删除的一个问题

以下代码错误：

```
vecotr<int>::iterator it = vc.begin();  
for( ; it != vc.end(); it++ )  
{   if( ***** )   vc.erase(it); }
```

原因：erase()删除元素后，**it失效**，并不是指向下一个元素

解决方案：

```
for(it=v1.begin(); it<v1.end(); ) {  
    if(*****) it=v1.erase(it);  
    else it++;  
}
```

在C++11标准中，erase() 会返回一个iterator，这个iterator指向了“当前删除元素的后继元素”

4、在vector上应用算法

常用算法

- ▶ 排序 `sort()`
- ▶ 查找 `find()`
- ▶ 替换 `replace()`
- ▶ 合并 `merge()`
- ▶ 反序 `reverse()`
- ▶ 统计 `count()`
- ▶ 其他等等算法

许多算法往往以迭代器作参数。比如排序和查找都需要两个迭代器参数（表示起始位置、终止位置）

有的算法返回一个迭代器。比如 `find` 算法，在容器中查找一个元素，并返回一个指向该元素的迭代器

算法主要在头文件 `<algorithm>` 和 `<numeric>` 中定义

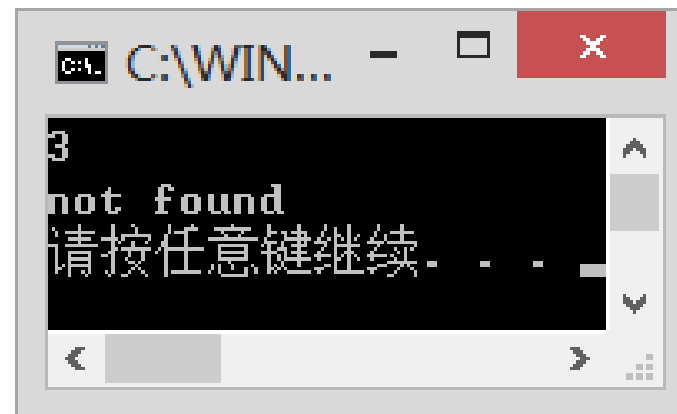
算法示例：find()

简化形式：find(first, last, val)

- ▶ first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找区间起点和终点。
 - 这个区间是个左闭右开的区间[first,last)，即区间的起点是位于查找范围之中的，而终点不是
- ▶ val参数是要查找的元素的值
- ▶ 函数返回值是一个迭代器。如果找到，则该迭代器指向被找到的元素。如果找不到，则该迭代器指向查找区间终点。

算法示例：find()

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main() {
    vector<int> v(5,3);
    vector<int>::iterator p;
    p = find(v.begin(),v.end(),3);
    if( p!=v.end()) cout<<*p<< endl;
    p = find(v.begin(),v.end(),5);
    if( p==v.end()) cout<<"not found\n";
    return 0;
}
```

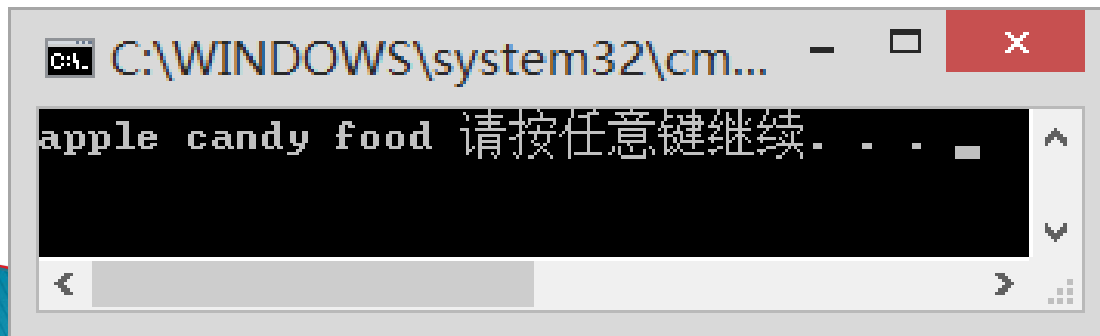


算法示例： sort ()

简化形式：

void sort(first, last)

- ▶ first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找区间起点和终点



```
C:\WINDOWS\system32\cm...
apple candy food 请按任意键继续. . .
```

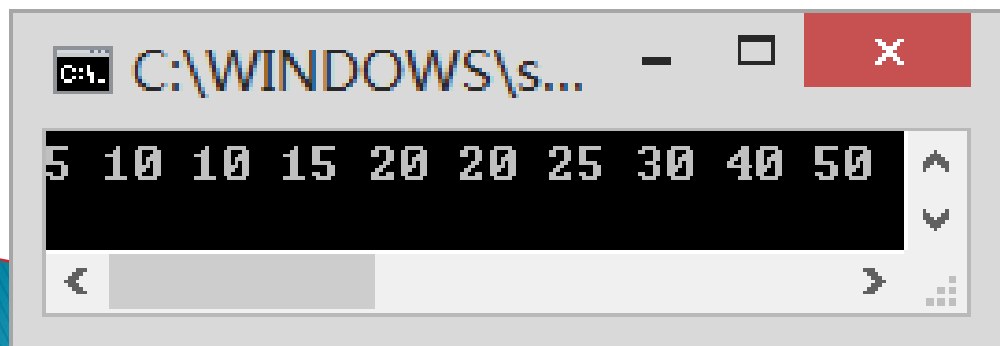
```
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;
int main() {
    vector<string> v;
    v.push_back("food");
    v.push_back("candy");
    v.push_back("apple");
    sort(v.begin(),v.end());
    vector<string>::iterator it;
    for(it=v.begin(); it!=v.end();it++)
        cout<< *it <<" ";
    return 0;
}
```

算法示例：merge()

形式：

merge(f1, e1, f2, e2, p)

- ▶ f1、e1、f2、e2、p都是迭代器
- ▶ 将有序序列v1中[f1, e1)和有序序列v2中[f2, e2)合并成有序序列，存入p的前面



```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main() {
    int A[] = {5, 10, 15, 20, 25};
    int B[] = {50, 40, 30, 20, 10};
    vector<int> v(10);
    vector<int>::iterator it;
    sort(A, A+5);    sort(B, B+5);
    merge(A, A+5, B, B+5, v.begin());
    for(it=v.begin(); it!=v.end(); ++it)
        cout<<*it<<" ";
    return 0;
}
```

其他算法示例

`replace(first, last, old, new)` // first, last为迭代器
作用：将 [first,last) 范围内的所有值为old的替换为new

`reverse(start, end)` // start, end为迭代器
作用：将序列中 [start, end) 范围反转排列顺序

`count(start, end, searchValue)` // start, end为迭代器
作用：统计[start, end) 范围内等于searchValue的元素个数

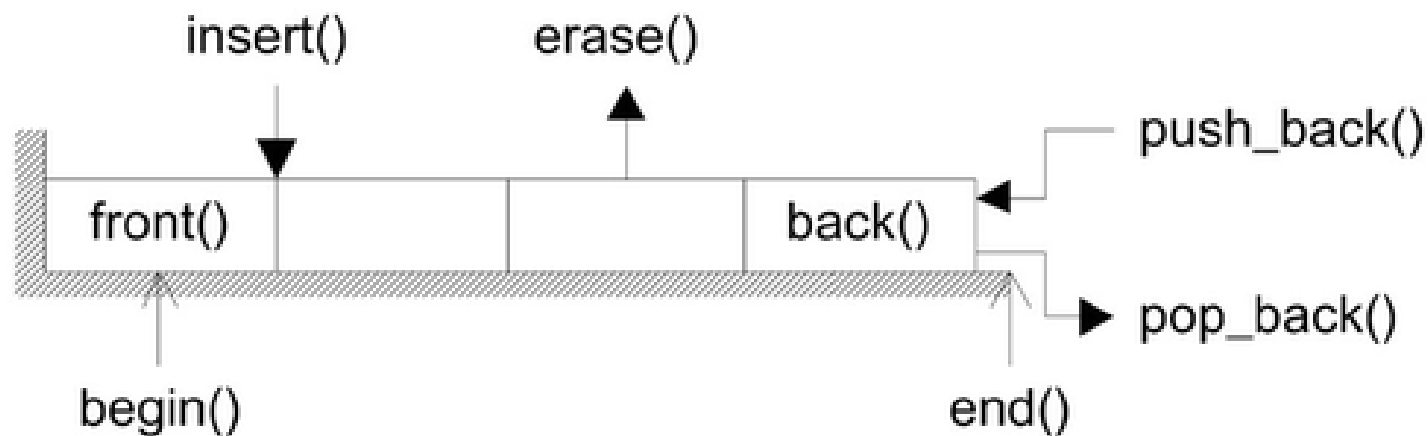
`accumulate(first, last, init)` // first, last为迭代器
作用：将[first,last)范围内的所有值相加，再加上init后返回

STL中算法众多，算法可能不一定适合的所有容器，使用时多查询手册

5、序列型容器概览

vector (向量)

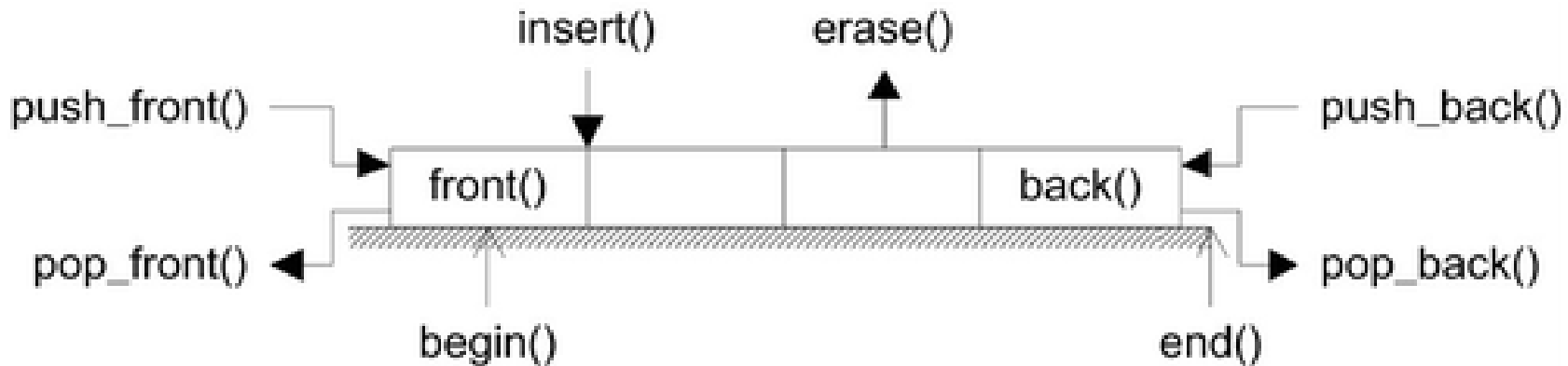
- 定义在头文件 `<vector>`
- 实际上就是个动态数组。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能。



向量 (vector)

deque（双端队列）

- 定义于头文件 <deque>
- 也是个动态数组，随机存取任何元素都能在**常数时间**完成(但性能次于vector)。在**两端增删元素**具有较佳的性能。

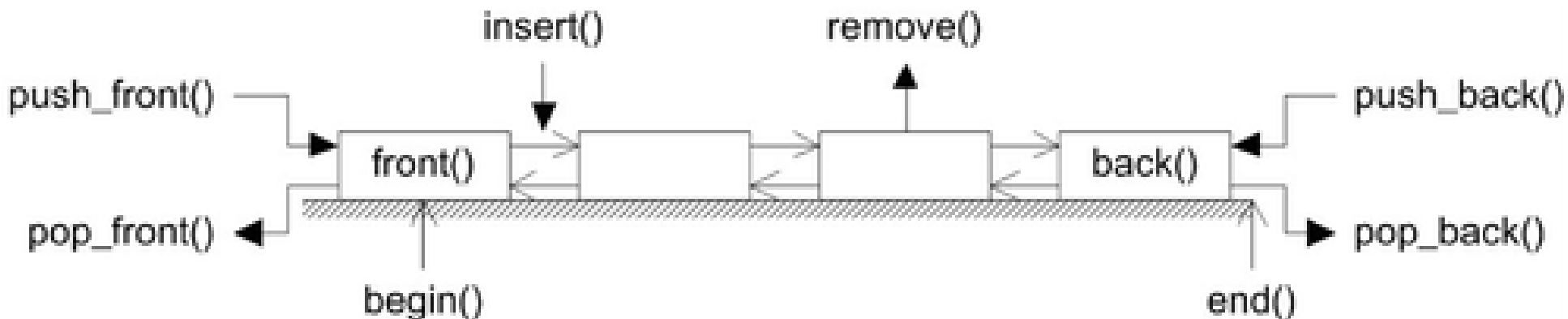


双端队列（deque）

list（双向链表）

- 定义于头文件 <list>
- 任意位置插入和删除元素的效率都很高
- 不支持随机存取
- 每个元素还有指针占用额外空间

在序列容器中，
元素的插入位置
同元素的值无关



列表（list）

序列容器初始化

- 默认构造函数初始化

```
vector<int> vec;  
list<string> list1;  
deque<float> deq;
```

- 拷贝构造函数初始化

```
vector<int> vec1;  
vector<int> vec2(vec1);  
list<string> list1;  
list<string> list2(list1);  
deque<float> deq1;  
deque<float> deq2(deq1);
```

- 创建有长度为10的容器

```
vector<string> vec(10);  
list<int> list1(10);  
deque<string> deq(10);
```

- 创建有10个初值的容器

```
vector<string> vec(10, "hi");  
list<int> list1(10, 1);  
deque<string> deq(10, "hi");
```

序列容器——添加元素

c.push_back(t)

在容器c的尾部添加值为t的元素。返回void类型

c.push_front(t)

在容器c的前端添加值为t的元素。返回void类型，只适用于list和deque

c.insert(p,t)

在迭代器p所指向的元素前面元素t。返回指向新添加元素的迭代器

c.insert(p,n,t)

在迭代器p所指向的元素前面插入n个值为t的新元素，返回void类型

c.insert(p,b,e)

在迭代器p所指向的元素前面插入迭代器b和e标记的范围内的元素。返回void类型

序列容器——访问元素

`c.back()`

返回容器c的最后一个元素的引用

`c.front()`

返回容器c的第一个元素的引用

`c[n]`

返回下标为n的元素的引用 ($0 \leq n < c.size()$) , 只适用于
vector和deque容器

`c.at[n]`

返回下标为n的元素的引用 ($0 \leq n < c.size()$) , 只适用于
vector和deque容器

序列容器——删除元素

c.pop_back()

删除容器c的最后一个元素

c.pop_front()

删除容器c的第一个元素，只适用于deque和list容器

c.erase(p)

删除迭代器p指向的容器中的元素

c.erase(b,e)

删除迭代器b和e所标记范围内的元素

c.clear()

删除容器中所有的元素

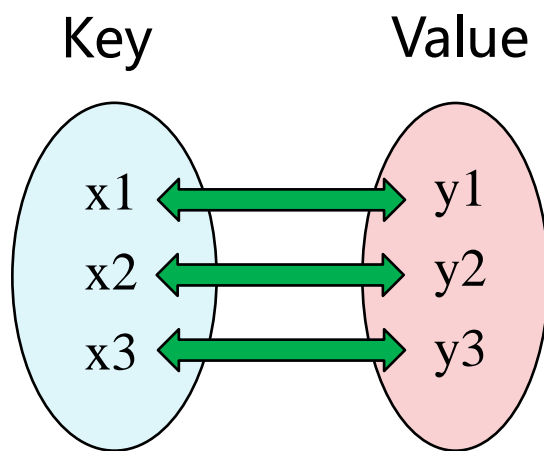
序列容器的选用

- 如果程序要求随机访问元素，则应用vector或者deque容器
- 如果程序必须在容器中间位置插入或删除元素，则应采用list容器
- 如果程序不是在容器的中间位置，而是在容器的首部或尾部插入或删除元素，则应采用deque容器

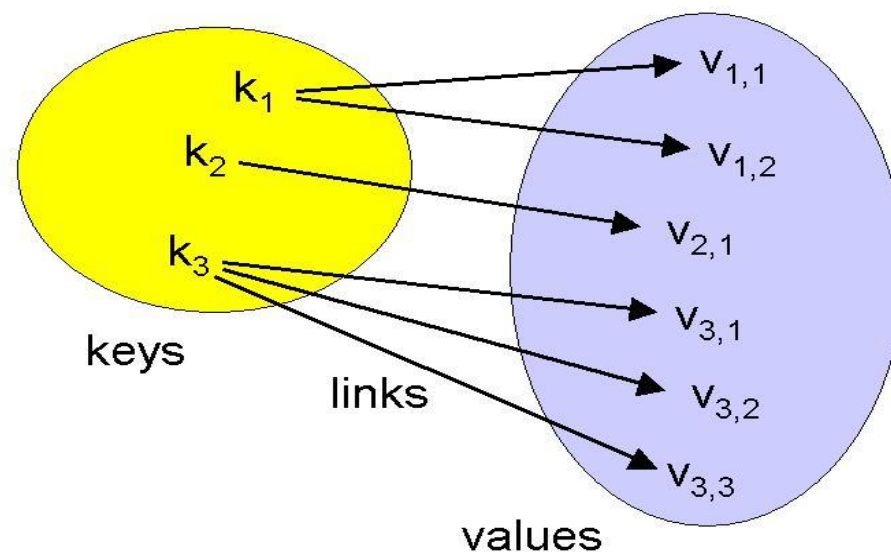
6、关联型容器概览

关联容器的特征

STL提供了4个关联容器，包括：map（映射）、multimap（多重映射）、set（集合）、multiset（多重集合）



map（映射）

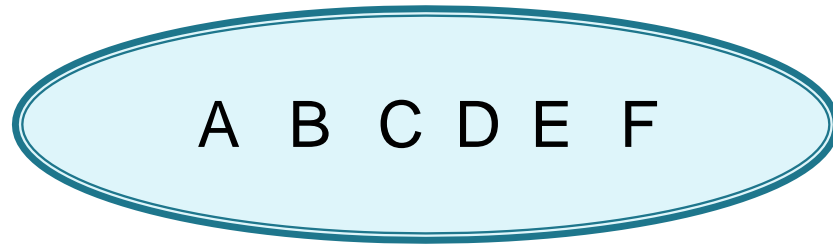


multimap（多重映射）

map、multimap的元素由（key，value）二元组构成，其中键必须是唯一的

关联容器的特征

- set 、 multiset 相当于只有键 (key) , 没有对应值 (value) 的 map 和 multimap
- set 支持通过键实现的快速读取 , 元素唯一



- multiset支持同一个键多次出现的set类型



关联容器与序列容器的差别

- 关联容器是通过**键(key)**存储和读取元素
- 顺序容器则通过元素在容器中的**位置顺序**存储和访问元素。

map和set的底层机制都是通过一种称为“红黑树”的数据结构存取数据，这使得它们的数据存取效率相当高

注：“红黑树”是一种常见的数据结构，感兴趣的同学可查看数据结构相关书籍

7、map容器初步

pair类型

- pair 类定义在 <utility> 头文件中。pair 是一个类模板，它将两个值组织在一起，这两个值的类型可不同。可以通过 first 和 second 公共数据成员来访问这两个值
- pair对象常常作为元素被添加到map中
- pair对象的定义：

```
pair<int, string> mypair(5 , "Jack"); //调用构造函数
```

```
pair<int, string> otherPair ; // 直接赋值
```

```
otherPair.first = 6;
```

```
otherPair.second = "Mike";
```

- 函数模板 make_pair() 能从两个变量构造一个 pair
- ```
pair<int, int > aPair = make_pair(5, 10);
```

# map创建及添加元素

➤ map 类定义在 <map> 头文件中

➤ 创建map对象：

```
map<int, string> StuInfo;
```

这就定义了一个用int作为键, 相关联string为值的map

➤ 插入pair对象：

```
pair<int, string> mypair(1, "Tom");
```

```
StuInfo.insert(mypair);
```

```
StuInfo.insert(pair<int, string>(5, "Jack"));
```

# map中使用运算符[ ]

- ▶ 用[ ]操作符修改元素的值 ( 键不可修改 )

```
StuInfo[1] = "Jim";
```

因为键为 1 的元素存在，因此修改元素

- ▶ 用[ ]操作符添加元素

```
StuInfo[2] = "Lily";
```

先查找主键为2的项，没找到，因此添加这个键为 2 的项

- ▶ 用[ ]取得元素的值

```
cout<<StuInfo[5]; // 输出键 5 对应的值
```

# 在map中查找元素

- 用find()查找map中是否包含某个关键字

```
int target = 3;
map<int,string>::iterator it;
it = StuInfo.find(target); //查找关键字target
if(it == StuInfo.end()){
 cout<<"not existed!"
}else{
 cout<<"find it!"<<endl;
}
```

若查找成功则返回目标项的迭代器，否则返回  
StuInfo.end() 迭代器

# 在map中删除元素

- 通过erase()函数按照关键字删除  
//删掉关键字"1"对应的条目  
`int r = StuInfo.erase(1);`  
若删除成功，返回 1 ，否则返回 0
- 用clear()清空map  
`StuInfo.clear();`



# 再论迭代器

- ▶ STL 中的迭代器按功能由弱到强分为5种：
  1. 输入：Input iterators 提供对数据的只读访问。
  1. 输出：Output iterators 提供对数据的只写访问
  2. 正向：Forward iterators 提供读写操作，并能一次一个地向前推进迭代器。
  3. 双向：Bidirectional iterators提供读写操作，并能一次一个地向前和向后移动。
  4. 随机访问：Random access iterators提供读写操作，并能在数据中随机移动。
- ▶ 编号大的迭代器拥有编号小的迭代器的所有功能，能当作编号小的迭代器使用。

# 不同迭代器所能进行的操作

- ▶ 所有迭代器:  $++p$ ,  $p++$
- ▶ 输入迭代器:  $*p$ ,  $p = p1$ ,  $p == p1$ ,  $p != p1$
- ▶ 输出迭代器:  $*p$ ,  $p = p1$
- ▶ 正向迭代器: 上面全部
- ▶ 双向迭代器: 上面全部,  $--p$ ,  $p--$ ,
- ▶ 随机访问迭代器: 上面全部, 以及:
  - $p += i$ ,  $p -= i$ ,
  - $p + i$  返回指向  $p$  后面的第  $i$  个元素的迭代器
  - $p - i$  返回指向  $p$  前面的第  $i$  个元素的迭代器
  - $p < p1$ ,  $p <= p1$ ,  $p > p1$ ,  $p >= p1$

# 容器所支持的迭代器类别

## 容器

vector

deque

list

set/multiset

map/multimap

stack

queue

## 迭代器类别

随机

随机

双向

双向

双向

不支持迭代器

不支持迭代器

关联容器支持双向迭代器，它支持：

\*、++、--、=、==、!=

不支持 <、<=、>=、>

# map中迭代器的使用

下面迭代器中”<”使用错误：

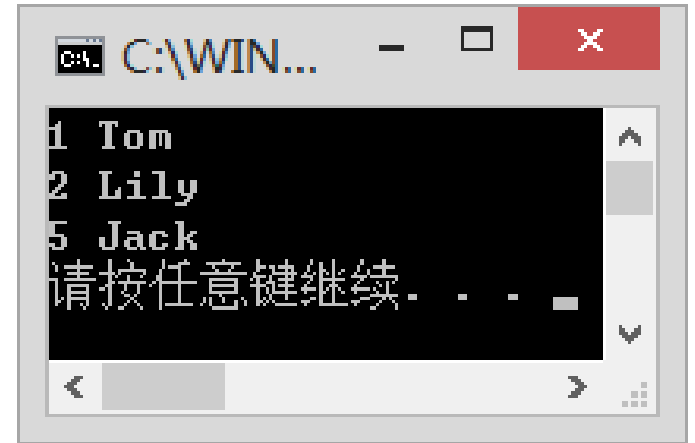
```
map<int,string> m;
map<int,string>::iterator it;
for(it = m.begin();it < m.end(); it++)
{ ***** }
```

下面是map迭代器正确的用法：

```
map<int,string> m;
map<int,string>::iterator it;
for(it = m.begin();it != m.end(); it++)
{ ***** }
```

# map中使用迭代器(例)

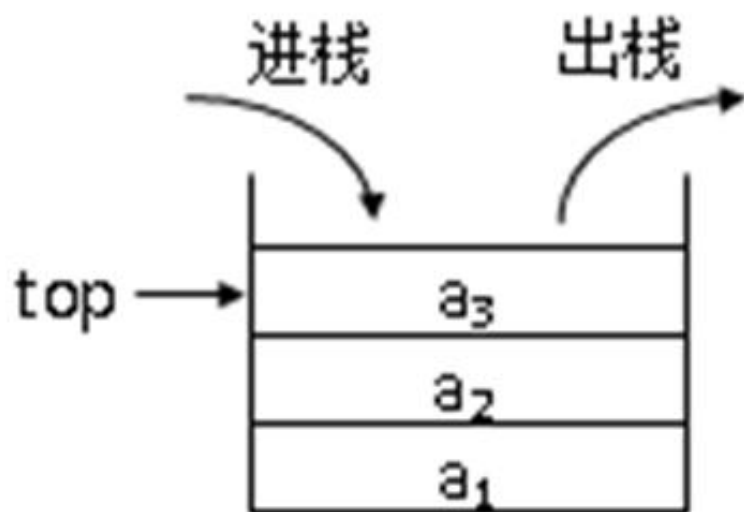
```
#include <iostream>
#include <string>
#include <utility>
#include <map>
using namespace std;
int main () {
 map<int,string> StuInfo;
 StuInfo.insert(pair<int, string>(1, "Tom"));
 StuInfo.insert(pair<int, string>(5, "Jack"));
 StuInfo[2]="Lily";
 map<int,string>::iterator it;
 for(it = StuInfo.begin();it != StuInfo.end(); it++)
 cout<<(*it).first<<" "<<(*it).second<<endl;
 return 0;
}
```



## 8、容器适配器概览

- 容器适配器将其他容器加以包装、改造，变成新的容器。实质上是一种受限容器
- 典型容器适配器：
  - stack（栈）
  - queue（队列）

# stack-堆栈



- 栈是限制在结构的一端进行插入和删除操作
- 允许进行插入和删除操作的一端称为栈顶，另一端称为栈底



# stack-堆栈

- 编程时加入下列语句：

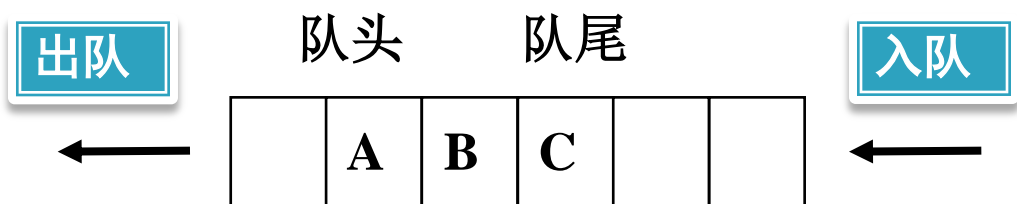
`#include <stack>`

- 栈的常用函数有：

|                         |           |
|-------------------------|-----------|
| <code>push(elem)</code> | 将元素elem入栈 |
| <code>pop()</code>      | 栈顶元素出栈    |
| <code>top()</code>      | 求栈顶元素     |
| <code>empty()</code>    | 判断栈是否空    |
| <code>size()</code>     | 求栈内元素个数   |

# queue-队列

只能在一端进行插入、在另一端进行删除操作的线性结构



队列示意图

➤ 加入下列语句：

```
#include <queue>
```

➤ 队列的常用函数有：

push()      入队

pop()      出队

front()      读取队首元素

back()      读取队尾元素

empty()      判断队列是否空

size()      求队列长度

# 堆栈示例

```
#include<iostream>
#include<stack>
using namespace std;
int main()
{
 stack<int> s; //定义栈 s
 s.push(1); s.push(2); s.push(3); s.push(9); //入栈
 cout<<"栈顶元素："<<s.top()<<endl; //读栈顶元素
 cout<<"元素数量："<<s.size()<<endl; //返回元素个数
 cout<<"出栈过程：";
 while(s.empty()!=true) //栈非空
 { cout<<s.top()<<" "; //读栈顶元素
 s.pop(); //出栈，删除栈顶元素
 }
 return 0;
}
```

栈顶元素： 9  
元素数量： 4  
出栈过程： 9 3 2 1