Practice of Programming 6 container

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

内容提纲

- 顺序容器
- 关联容器

什么是顺序容器

- 顺序容器：sequence container
  - 拥有由单一类型元素组成的一个有序集合
- 两个主要的顺序容器是list 和vector
- 第三个顺序容器为双端队列deque
  - 提供了与vector 相同的行为，但是对于首元素的有效插入和删除提供了特殊的支持。

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

示例系统简介-文本查询系统

- 功能：
  - 用户指定一个任意的文本文件
  - 可以查询其中的单词或相邻的单词序列
  - 显示要查找的单词或单词序列的出现次数
  - 还可以把包含单词或单词序列的句子显示出来

- 如果用户希望找到所有对Civil War 或Civil Rights 的引用则查询可能如下
  - Civil && (War || Rights)
- 查询结果如下
  - Civil: 12 occurrences
  - War: 48 occurrences
  - Rights: 1 occurrence
  - Civil && War: 1 occurrence
  - Civil && Rights: 1 occurrence
  - (8) Civility, of course, is not to be confused with Civil Rights, nor should it lead to Civil War.

示例系统简介-文本查询系统

- 任务：
  1. 它必须允许用户指明要打开的文本文件的名字
  2. 它必须在内部组织文本文件，以便能够识别出每个单词在句子中出现的次数，以及在该句子中的位置
  3. 它必须支持某种形式的布尔查询语言，在该系统中将支持
     - && 在一行中两个单词不仅存在而且相邻
     - || 在一行中两个单词至少有一个存在
     - ! 在一行中该单词不存在
     - () 把子查询组合起来的方式

示例系统简介-文本查询系统

- 文本：

  Alice Emma has long flowing red hair. Her Daddy says when the wind blows through her hair, it looks almost alive, like a fiery bird in flight. A beautiful fiery bird, he tells her, magical but untamed, "Daddy, shush, there is no such thing," she tells him, at the same time wanting him to tell her mere. Shyly, she asks, "I mean, Daddy, is there?"

- 存储：
  - 读入文本的每一行
  - 把它们分成独立的单词
  - 去掉标点符号
  - 把大写字母变成小写
  - 提供关于后缀的最小支持
  - 去掉无语义的词比如and 、a 和the

示例系统简介-文本查询系统

alice ((0,0))
alive ((1,10))
almost ((1,9))
ask ((5,2))
beautiful ((2,7))
bird ((2,3),(2,9))
blow ((1,3))
……
tell ((2,11),(4,1),(4,10))
there ((3,5),(5,7))
thing ((3,9))
through ((1,4))
time ((4,6))
untamed ((3,2))
wanting ((4,7))
wind ((1,2))

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

vector和list

- vector
  - 表示一段连续的内存区域，每个元素被顺序存储在这段内存中
  - 随机访问效率高 vs. 任意位置插入与删除效率低
- deque
  - 也表示一段连续的内存区域，但是与vector 不同的是：它支持高效地在其首部插入和删除元素
- List
  - 表示非连续的内存区域，并通过一对指向首尾元素的指针双向链接起来从而允许向前和向后两个方向进行遍历
  - 随机访问效率低 vs. 任意位置插入与删除效率高
  - 每个元素还有两个指针的额外空间开销

vector和list

- 下面是选择顺序容器类型的一些准则
  - 需要随机访问一个容器，vector ＞list
  - 已知要存储元素的个数，vector ＞list
  - 需要的不只是在容器两端插入和删除元素，list ＞vector
  - 除非需要在容器首部插入和删除元素，
    否则vector ＞deque

- 既需要随机访问元素又需要随机插入和删除元素，那么又该怎么办呢？

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

vector自增长

- vector
  - 分配了一些额外的内存容量或者说它预留了这些存储区
  - 分配的额外容量的确切数目由具体实现定义
  - 对于小的对象，vector 在实践中比list效率更高

- 容量和长度
  - 容量(capacity)是指在容器下一次需要增长自己之前能够被加入到容器中的元素的总数
    - capacity()操作
  - 长度(size)是指容器当前拥有元素的个数
    - size()操作

vector自增长


```
#include <vector>
#include <iostream>
int main()
{
    vector< int > ivec;
    cout << "ivec: size: " << ivec.size()
         << " capacity: " << ivec.capacity() << endl;
    for ( int ix = 0; ix < 24; ++ix ) {
        ivec.push_back( ix );
        cout << "ivec: size: " << ivec.size()
             << " capacity: " << ivec.capacity() << endl;
    }
}
```

- ivec初始值为0，放入第一个元素后变成256
- 以后每当容量满时，容量翻倍，并将原内存拷贝至新内存，然后释放原内存

vector自增长

数据类型	长度（字节）	初始插入后的容量
int	4	256
double	8	128
简单类	12	85
String	12	85
大型简单类	8000	1
大型复杂类	8000	1

- 这是标准库的Rogue Wave 实现版本

vector自增长

数据类型	list	vector
int	1038	3.76
double	10.72	3.95
简单类	12.31	5.89
String	14.42	11.80

- 插入1 千万个元素所需的时间

数据类型	list	vector
大型简单类	0.36	2.23
大型复杂类	2.37	6.70

- 插入1 万个元素的时间

vector自增长

- 简单类对象和大型简单类对象：
  - 通过按位拷贝插入，即一个对象的所有位被拷贝到第二个对象的位中
- string 类对象和大型复杂类对象：
  - 通过调用拷贝构造函数来插入
- vector 的动态自增长越频繁，元素插入开销就越大
  - 把vector转换为list
  - 用指针类间接存储复杂类的对象

vector自增长

- reserve()：显式地设置容器的容量

```
int main() {
    vector< string > svec;
    svec.reserve( 32 )  // 把容量设置为32
    // ...
}
```

- 简单类 vs. 大型复杂类

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

定义一个顺序容器

- 必须包含下列头文件之一

```
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
```

- 容器类型的名字开始，后面是所包含的元素的实际类型

```
vector< string > svec;
list< int > ilist;
```

定义一个顺序容器

- 常用操作方法

empty()　　判断是否为空

```
if ( svec.empty() != true )
```

假设有：

push_back()　将元素插入在容器的尾部

```
for ( int ix = 0; ix < 4; ++ix )
    ilist.push_back( ia[ ix ] );
```

列表中元素的顺序是：0 1 2 3

push_front()　把新元素插入在链表的前端（list　deque）

```
for ( int ix = 0; ix < 4; ++ix )
    ilist.push_front( ia[ ix ] );
```

列表中元素的顺序是：3 2 1 0

定义一个顺序容器

- 指定长度：常量或变量

```
extern int get_word_count( string file_name );
const int list_size = 64;
list< int > ilist( list_size );
vector< string > svec(get_word_count(string("Chimera")));
```

- 调整长度

```
svec.resize( 2 * svec.size() );
```

定义一个顺序容器

- 初始化

  缺省初始化：初始化为与该类型相关联的缺省值

  整数:用0初始化所有元素

  string 类每个元素都将用string 的缺省构造函数初始化

  指定初始化：

```
list< int > ilist( list_size, - 1 );
 vector< string > svec( 24, "pooh" );
```

  调整尺寸后初始化

```
svec.resize( 2 * svec.size(), "piglet" );
```

  用现有容器对象初始化新的容器对象

```
 vector< string > svec2( svec );
 list< int > ilist2( ilist );
```

定义一个顺序容器

- 容器比较
  - 容器的比较是指两个容器的元素之间成对进行比较
  - 如果所有元素相等且两个容器含有相同数目的元素则两个容器相等
  - 否则它们不相等，且第一个不相等元素的比较决定了两个容器的小于或大于关系

```
ivec1: 1 3 5 7 9 12
ivec2: 0 1 1 2 3 5 8 13
ivec3: 1 3 9
ivec4: 1 3 5 7
ivec5: 2 4
// 第一个不相等元素: 1, 0
// ivec1 大于 ivec2
ivec1 < ivec2 // flase
ivec2 < ivec1 // true
// 第一个不相等元素: 5, 9
ivec1 < ivec3 // true
// 所有元素相等, 但是, ivec4 的元素少
// 所以ivec4 小于ivec1
ivec1 < ivec4 // false
// 第一个不相等元素: 1, 2
ivec1 < ivec5 // true
ivec1 == ivec1 // true
ivec1 == ivec4 // false
ivec1 != ivec4 // true
ivec1 > ivec2 // true
ivec3 > ivec1 // true
ivec5 > ivec2 // true
```

定义一个顺序容器

- 自定义容器类型

  三个限制：
  - 元素类型必须支持"等于"操作符
  - 元素类型必须支持"小于"操作符
  - 元素类型必须支持一个缺省值（对于类类型即指缺省构造函数）

- 所有预定义数据类型，包括指针，都满足这些限制
- C++标准库给出的所有类类型也一样

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

迭代器

- 迭代器（iterator ）
  - 提供了一种对顺序或关联容器类型中的每个元素进行连续访问的通用方法
  - 假设iter 为任意容器类型的一个iterator， 则

    ++iter;　向前移动迭代器使其指向容器的下一个元素

    *iter;　　返回iterator 指向元素的值
  - 每种容器类型都提供：
    - begin()成员函数，返回指向容器的第一个元素的iterator
    - end()函数，返回指向容器的末元素的下一个位置的iterator

迭代器

- 定义迭代器：

```
vector<string> vec;
vector<string>::iterator iter = vec.begin();
vector<string>::iterator iter_end = vec.end();

for( ; iter != iter_end; ++iter )
    cout << *iter << '\n';

for ( iter = container.begin();iter != container.end(); ++iter )
  do_something_with_element( *iter );
```

迭代器

- const_iterator 类型：以只读方式遍历const 容器

```
#include <vector >
void even_odd( const vector<int> *pvec,
                    vector<int> *pvec_even,
                    vector<int> *pvec_odd )
{
    // must declare a const_iterator to traverse pvec
    vector<int>::const_iterator c_iter = pvec->begin();
    vector<int>::const_iterator c_iter_end = pvec->end();
    for ( ; c_iter != c_iter_end; ++c_iter )
      if ( *c_iter % 2 )
                pvec_odd->push_back( *c_iter );
      else pvec_even->push_back( *c_iter );
```

迭代器

- 用标量算术运算来移动迭代器的位置

```
vector<int>::iterator iter = vec.begin()+vec.size()/2;
iter += 2;
```

- iterator 算术运算
  - 只适用于vector 或deque
  - 不适用于list
  - 因为list 的元素在内存中不是连续存储的

迭代器

- 用迭代器来初始化容器

```
#include <vector>
#include <string>
#include <iostream>
int main()
{
    vector<string> svec;
    string intext;
    while ( cin >> intext )
    svec.push_back( intext );
    // 用svec 的全部元素初始化svec2
    vector<string> svec2( svec.begin(), svec.end() );
    // 用svec 的前半部分初始化svec3
    vector<string>::iterator it = svec.begin() + svec.size()/2;
    vector<string> svec3( svec.begin(), it );
    // 处理 vectors ...
}
```

迭代器

- 指向内置数组的指针也可以被用作元素范围标记器

```
#include <string>
string words[4] = {
    "stately", "plump", "buck", "mulligan"
};
vector< string > vwords( words, words+4 );

int ia[6] = { 0, 1, 2, 3, 4, 5 };
list< int > ilist( ia, ia+6 );
```

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

顺序容器操作


```
vector< string > svec;
list< string > slist;
string spouse( "Beth" );
slist.insert( slist.begin(), spouse );
svec.insert( svec.begin(), spouse );
```

- insert()　第一种形式
  - 第一个参数是一个位置指向容器中某个位置的iterator
  - 第二个参数是将要被插入的值这个值被插入到由iterator 指向的位置的前面

顺序容器操作


```
string son( "Danny" );
list<string>::iterator iter;
iter = find( slist.begin(), slist.end(), son );
slist.insert( iter, spouse );
```

- find()
  - 返回被查找元素在容器中的位置
  - 或者返回容器的end() iterator 表明这次查询失败

- push_back()等价于:

```
slist.push_back( value ); ⇔ slist.insert( slist.end(), value );
```

顺序容器操作

- insert() 第二种形式
  - 支持在某个位置插入指定数量的元素

```
vector<string> svec;
...
string anna( "Anna" );
svec.insert( svec.begin(), 10, anna );
```

- insert() 第三种形式
  - 支持向容器插入一段范围内的元素

```
string sarray[4] = { "quasi", "simba", "frollo", "scar" };
```

顺序容器操作

- 向字符串vector 中插入数组中的全部或部分元素

```
svec.insert( svec.begin(), sarray, sarray+4 );
svec.insert( svec.begin() + svec.size()/2, sarray+2, sarray+4);
```

- 可以通过一对iterator 来标记出待插入值的范围

```
// 插入svec 中含有的元素
// 从svec_two 的中间开始
svec_two.insert( svec_two.begin() + svec_two.size()/2,
svec.begin(), svec.end() );
```

顺序容器操作

- 更一般的情况下可以是任意一种string 对象的容器

```
list< string > slist;
// ...
// 把svec 中含有的元素插入到
// slist 中stringVal 的前面
list< string >::iterator iter =
    find( slist.begin(), slist.end(), stringVal );
slist.insert( iter, svec.begin(), svec.end() );
```

删除操作

- erase()：第一种形式
  - 删除单个元素

```
string searchValue( "Quasimodo" );
list< string >::iterator iter =
    find( slist.begin(), slist.end(), searchValue );
if ( iter != slist.end() )
    slist.erase( iter );
```

删除操作

- erase()：第二种形式
  - 删除由一对iterator 标记的一段范围内的元素

```
// 删除容器中的所有元素
slist.erase( slist.begin(), slist.end() );
// 删除由iterator 标记的一段范围内的元素
list< string >::iterator first, last;
first = find( slist.begin(), slist.end(), val1 );
last = find( slist.begin(), slist.end(), va
// ... 检查first 和last 的有效性
slist.erase( first, last );
```

删除操作

- pop_back()方法：删除容器的末元素
  - 它不返回元素只是简单地删除它

```
vector< string >::iterator iter = buffer.begin();
for ( ; iter != buffer.end(); iter++ )
{
    slist.push_back( *iter );
    if ( ! do_something( slist ))
      slist.pop_back();
}
```

赋值与对换

- 赋值操作符
  - 使用针对容器元素类型的赋值操作符，把右边容器对象中的元素依次拷贝到左边的容器对象。
  - 它不返回元素只是简单地删除

```
// slist1 含有10 个元素
// slist2 含有24 个元素
// 赋值之后都含有24 个元素
slist1 = slist2;
```

赋值与对换

- 对换
  - swap()可以被看作是赋值操作符的互补操作

    slist1.swap(slist2);
  - slist1 现在含有24 个string 元素，是用string 赋值操作符拷贝的

- 赋值与对换的区别：
  - 在于slist2 现在含有slist1 中原来含有的10 个元素的拷贝
  - 如果两个容器的长度不同则容器的长度就被重新设置且等于被对换容器的长度

– next

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

存储文本行

- 把每行拆成独立的单词
  - 首先去掉标点符号

For every tale there's a telling, and that's the he and she of it.“

"For there's telling, that's it."

For there telling that it

  - 丢弃没有语义的单词，如 is that and it the tale telling
  - 去掉大写字母

Home home

  - 处理后缀

love loves loving

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

寻找子串

- string 类提供一套查找函数，都以find 的各种变化形式命名
  - find()是最简单的实例：
    - 给出一个字符串，它返回匹配子串的第一个字符的索引位置，或者返回一个特定的值 string::npos表明没有匹配

```
#include <string>
#include <iostream>
int main() {
    string name( "AnnaBelle" );
    int pos = name.find( "Anna" );
    if ( pos == string::npos )
            cout << "Anna not found!\n";
    else cout << "Anna found at pos: " << pos << endl;
}
```

    - 返回的索引类型更严格的类型应该使用以下形式

```
string::size_type
string::size_type pos = name.find( "Anna" );
```

寻找子串

- find_first_of()：
  - 查找与被搜索字符串中任意一个字符相匹配的第一次出现
  - 返回它的索引位置

```
#include <string>
#include <iostream>
int main() {
    string numerics( "0123456789" );
    string name( "r2d2" );
    string::size_type pos = name.find_first_of( numerics );
    cout << "found numeric at index: "
         << pos << "\telement is “
         << name[pos] << endl;
}
```

寻找子串

- find_first_of()：
  - 第二个参数指明了字符串中起始查找位置的索引

```
#include <string>
#include <iostream>
int main() {
  string numerics( "0123456789" );
  string name( "r2d2" );
  string::size_type pos = 0;
  // 这里存在错误!
  while (( pos = name.find_first_of( numerics, pos ))
          != string::npos )
   cout << "found numeric at index: "
       << pos << "\telement is "
       << name[pos] << endl;
}
```

```
#include <string>
#include <iostream>
int main() {
  string numerics( "0123456789" );
  string name( "r2d2" );
  string::size_type pos = 0;
  while (( pos = name.find_first_of( numerics, pos ))
          != string::npos )
  {
   cout << "found numeric at index: "
       << pos << "\telement is "
       << name[pos] << endl;
     // 移到被找到元素的后一位置
   ++pos;
 }
}
```

寻找子串

- find_first_of()：
  - 为了标记出每个单词的长度我们引入了第二个位置对象

```
#include <string>
#include <iostream>
int main() {
  string numerics( "0123456789" );
  string name( "r2d2" );
 // pos: 单词后一位置的索引
 // prev_pos: 单词开始的索引
  string::size_type pos = 0, prev_pos = 0;
  while (( pos = name.find_first_of( numerics, pos ))
          != string::npos )
  {
   cout << "found numeric at index: "
        << pos << "\telement is "
        << name[pos] << endl;
    prev_pos = ++pos;
 }

pos - prev_pos: // 标识单词长度
```

```
#include <string>
#include <iostream>
int main() {
  string numerics( "0123456789" );
  string name( "r2d2" );
  string::size_type pos = 0;
  while (( pos = name.find_first_of( numerics, pos ))
          != string::npos )
  {
   cout << "found numeric at index: "
        << pos << "\telement is "
        << name[pos] << endl;
      // 移到被找到元素的后一位置
   ++pos;
 }
}
```

寻找子串

- substr() :
  - 生成现有string 对象的子串的一个拷贝
  - 第一个参数指明开始的位置
  - 第二个可选的参数指明子串的长度，如果省略第二个参数，将拷贝字符串的余下部分

```
vector<string> words;
while (( pos = textline.find_first_of( ' ', pos ))
                != string::npos )
{
    words.push_back( textline.substr(prev_pos, pos-prev_pos));
    prev_pos = ++pos;
}
```

寻找子串

- 下面是完整的实现，已经放入一个被称为separate_words()的函数中。
- 除了把每个单词存储在字符串vector 中之外，还计算了每个单词的行列位置

```
    string::size_type pos = 0, prev_pos = 0;
    while (( pos = textline.find_first_of( ' ', pos ))
         != string::npos )
    {
       // 存储当前单词子串的拷贝
       words->push_back(textline.substr( prev_pos, pos - prev_pos ));
       // 将行/列信息存储为pair
       locations ->push_back(make_pair( line_pos, word_pos ));
       // 为下一次迭代修改位置信息
       ++word_pos;
       prev_pos = ++pos;
    }
    // 现在处理最后一个单词
    words->push_back(textline.substr( prev_pos, pos - prev_pos ));
    locations->push_back(make_pair( line_pos, word_pos ));
  }
  return new text_loc( words, locations );
}
```

寻找子串

- separate_words()在text_file 上的部分执行情况

```
eol: 52 pos: 5 line: 0 word: 0 substring: Alice
eol: 52 pos: 10 line: 0 word: 1 substring: Emma
eol: 52 pos: 14 line: 0 word: 2 substring: has
eol: 52 pos: 19 line: 0 word: 3 substring: long
eol: 52 pos: 27 line: 0 word: 4 substring: flowing
eol: 52 pos: 31 line: 0 word: 5 substring: red
eol: 52 pos: 37 line: 0 word: 6 substring: hair.
eol: 52 pos: 41 line: 0 word: 7 substring: Her
eol: 52 pos: 47 line: 0 word: 8 substring: Daddy
last word on line substring: says
...
textline: magical but untamed. "Daddy, shush, there is no such thing,"
eol: 60 pos: 7 line: 3 word: 0 substring: magical
eol: 60 pos: 11 line: 3 word: 1 substring: but
eol: 60 pos: 20 line: 3 word: 2 substring: untamed.
eol: 60 pos: 28 line: 3 word: 3 substring: "Daddy,
eol: 60 pos: 35 line: 3 word: 4 substring: shush,
eol: 60 pos: 41 line: 3 word: 5 substring: there
eol: 60 pos: 44 line: 3 word: 6 substring: is
```

寻找子串

- rfind()：
  - 查找最后即最右的指定子串的出现
- find_first_not_of()：
  - 查找第一个不与要搜索字符串的任意字符相匹配的字符
- find_last_of()
  - 查找字符串中的与搜索字符串任意元素相匹配 的最后一个字符
- find_last_not_of()
  - 查找字符串中的与搜索字符串任意字符全不匹配的最后一个字符

- 这些操作都有一个可选的第二参数来指明起始的查找位置

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

处理标点符号

- 怎样去掉不想要的标点呢？
  - 定义一个字符串，它包含我们希望去掉的所有标点

```
string filt_elems( "\",.;:!?)(\\/" );
```

  - 用find_first_of()操作在字符串里找到每个匹配元素

```
while (( pos = word.find_first_of( filt_elems, pos ))
    != string::npos )
```

  - 需要用erase()去掉字符串中的标点

```
word.erase(pos, 1);
```

- erase()：删除
  - 第一个参数表示字符串中要被删除的字符的开始位置
  - 第二个参数是可选的表示要被删除的字符的个数

处理标点符号

- 下面是filter_text()的完整实现它有两个参数：
  - 指向包含文本的string vector 的指针
  - 以及含有要过滤的元素的string 对象

```
void  filter_text( vector<string> *words, string filter )
{
        vector<string>::iterator iter = words ->begin();
        vector<string>::iterator iter_end = words ->end();
        // 如果用户没有提供filter, 则缺省使用最小集
        if ( ! filter.size() )
                filter.insert( 0, "\".," );
        while ( iter != iter_end ) {
                string::size_type pos = 0;
                // 对于找到的每个元素, 将其删除
                while (( pos = (*iter).find_first_of( filter, pos ))
                        != string::npos )
                    (*iter).erase(pos,1);
                iter++;
        }
```

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

任意其他格式的字符串

- 解决大小写字母

```
void strip_caps( vector<string,allocator> *words )
{
    vector<string,allocator>::iterator iter = words ->begin();
    vector<string,allocator>::iterator iter_end = words ->end();
    string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
    while ( iter != iter_end ) {
      string::size_type pos = 0;
      while (( pos = (*iter).find_first_of( caps, pos ))
                    != string::npos )
        (*iter)[ pos ] = tolower( (*iter)[pos] );
      ++iter;
    }
}
```

- tolower( )：#include <ctype.h>
  - 是标准C 库函数，接受一个大写字符并返回与之等价的小写字母

任意其他格式的字符串

- 处理以s 结尾的单词

```
void suffix_text( vector<string,allocator> *words )
{
    vector<string,allocator>::iterator
    iter = words->begin(),
    iter_end = words->end();
    while ( iter != iter_end ) {
      // 如果只有3 个字符或者更少则不加处理
      if ( (*iter).size() <= 3 )
            { ++iter; continue; }
      if ( (*iter)[ (*iter).size ()-1 ] == 's' )
            suffix_s( *iter );
      // 其他后缀的处理比如ed ing ly 等
      ++iter;
    }
}
```

任意其他格式的字符串

- 处理以s 结尾的单词

```
string::size_type spos = 0;
string::size_type pos3 = word.size()-3;
// "ous", "ss", "is", "ius"
string suffixes( "oussisius" );
if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) || // ous
     ! word.compare( pos3, 3, suffixes, spos+6, 3 ) || // ius
     ! word.compare( pos3+1, 2, suffixes, spos+2, 2 ) || // ss
     ! word.compare( pos3+1, 2, suffixes, spos+4, 2 )) // is
return;
```

- 如果两个字符串的比较结果相等则compare()返回0

本节内容

- 示例系统简介
- vector和list
- vector自增长
- 定义一个顺序容器
- 迭代器
- 顺序容器操作
- 存储文本行
- 寻找子串
- 处理标点符号
- 任意其他格式的字符串
- 其他string操作

其他string操作

- erase()的第二种形式
  - 用一对迭代器iterator 作参数标记出要删除的字符范围

    name.erase(name.begin()+startPos,name.begin()+endPos);
- erase()的第三种形式
  - 只带一个iterator 作参数，标记出要被删除的字符

    name.erase(endPos);

- insert()操作支持将另外的字符插入到指定的位置

  string_object.insert(position, new_string);
- insert()操作也支持插入new_string 的一个子部分

```
string_object.insert(
  string_object.size(), // string_object 中的位置
  new_string, pos, // new_string 的开始位置
  posEnd-pos // 要拷贝字符的数目
)
```

其他string操作

- assign()
  - 对字符串赋值
- append()
  - 在字符串末尾追加
- swap()
  - 操作交换两个string 的值
- at()
  - 提供运行时刻对索引值的范围检查
- compare()
  - 两个字符串的字典序比较
- replace()
  - 提供十种方式用一个或多个字符替换字符串中的一个或多个现有的字符

内容提纲

- 顺序容器
- 关联容器

什么是关联容器

- 关联容器：associative container
  - 支持查询一个元素是否存在并且可以有效地获取元素
- 两个主要的关联容器是map 和set
- multimap 和multiset 支持同一个键的多次出现

本节内容

- 生成文本位置map
- 创建单词排除集
- 完整的程序
- multimap 和multiset
- 栈
- 队列和优先级队列

生成文本位置map

- map 也叫关联数组associative array 中
  - 存储键/值对
  - 键用来索引map
  - 值用作被存储和检索的数据

- 为了使用map 我们必须包含相关的头文件

```
#include <map>
```

- 在使用map 和set 时两个最主要的动作是向里面放入元素以及查询元素是否存在

定义并生成map

- 定义map 对象我们至少要指明键和值的类型

```
map<string, int> word_count;

class employee;
map<int, employee*> personnel;
```

- 文本查询系统map 声明如下

```
typedef pair<short,short> location;
typedef vector<location> loc;
map<string,loc*> text_map;
```

定义并生成map

- 定义了map 以后下一步工作就是加入键/值元素对

```
#include <map>
#include <string>
map<string,int> word_count;
word_count[ string("Anna") ] = 1;
word_count[ string("Danny") ] = 1;
word_count[ string("Beth") ] = 1;
```

定义并生成map

- 一种比较好的插入单个元素的方法

```
typedef map<string,int>::value_type valType;

// the preferred single element insertion method
word_count.insert(
      map<string,int>::value_type( string("Anna"), 1 )
);
```

- 另一个版本的insert()方法：插入一定范围内的键/值元素

```
map< string, int > word_count;
// ... fill it up
map< string, int > word_count_two;
// 插入所有键/值对
word_count_two.insert(word_count.begin(),word_count.end());
map< string, int > word_count_two( word_count );
```

定义并生成map

- separate_words()创建了两个vector
  - 文本中所有词的字符串vector：为文本map 提供了键的集合
  - 相应的行列对的位置vector：为文本map提供了相关联的值集合

- separate_words()返回一个pair 对象，它拥有指向这两个vector 的指针，该pair 对象是函数build_word_map()的参数

```
// typedefs to make declarations easier
typedef pair< short,short > location;
typedef vector< location > loc;
typedef vector< string > text;
typedef pair< text*,loc* > text_loc;
extern map< string, loc* >*
    build_word_map( const text_loc *text_locations );
```

定义并生成map

- 准备工作

```
map<string,loc*> *word_map = new map< string, loc* >;
vector<string> *text_words = text_locations ->first;
vector<location> *text_locs = text_locations ->second;
```

定义并生成map

- 并行迭代两个vector
  - map 中还没有单词在这种情况下，需要插入键/值对
  - 单词已经被插入在这种情况下，需要用另外的行列信息修改该项的位置vector

```
register int elem_cnt = text_words ->size();
for ( int ix = 0; ix < elem_cnt; ++ix )
{
        string textword = ( *text_words )[ ix ];
        // 排除策略: 如果少于3 个字符,
        // 或在排除集合中存在,
        // 则不输入到map 中.
        if ( textword.size() < 3 || exclusion_set.count( textword ))
            continue;
      // 判断单词是否存在
      // 如果count()返回0 则不存在——加入它
        if ( ! word_map->count((*text_words)[ix] ))
        {
          loc *ploc = new vector<location>;
          ploc->push_back( (*text_locs)[ix] );
```

定义并生成map

- 下面给出怎样在main()函数中调用它

```
int main()
{
    // 读入并分离文本
    vector<string,allocator> *text_file = retrieve_text();
    text_loc *text_locations = separate_words( text_file);
    // 处理单词
    // ...

    // 生成单词/位置对并提示查询
    map<string,loc*,less<string>,allocator>
    *text_map = build_word_map( text_locations );
    // ...
}
```

查找并获取map 中的元素

- 下标操作符

```
map<string,int> word_count;
int count = word_count[ "wrinkles" ];
```

- Count(keyValue)：返回map 中keyValue 出现的次数

```
int count = 0;
if ( word_count.count( "wrinkles" ))
        count = word_count[ "wrinkles" ];
```

- Find(keyValue)：如果实例存在，则find()返回指向该实例的iterator，如果不存在，则返回等于end()的iterator

```
int count = 0;
map<string,int>::iterator it = word_count.find( "wrinkles" );
if ( it != word_count.end() )
        count = (*it).second;
```

对map 进行迭代


```
display_map_text()
display_map_text( map<string,loc*> *text_map )
{
    typedef map<string,loc*> tmap;
    tmap::iterator iter = text_map->begin(),
    iter_end = text_map->end();
    while ( iter != iter_end )
    {
       cout << "word: " << (*iter).first << " (";
       int loc_cnt = 0;
       loc *text_locs = (*iter).second;
       loc::iterator liter = text_locs->begin(),
       liter_end = text_locs->end();
       while ( liter != liter_end )
       {
          if ( loc_cnt )
             cout << ',';
          else ++loc_cnt;
          cout << '(' << (*liter).first<< ',' << (*liter).second << ')';
          ++liter;
       }
```

查找并获取map 中的元素

- size()函数

```
if ( text_map->size() )
display_map_text( text_map );
```

- empty()函数

```
if ( ! text_map->empty() )
display_map_text( text_map );
```

单词转换map

- 下面的小程序说明了怎样创建查找和迭代一个map

```
#include <map>
#include <vector>
#include <iostream>
#include <string>
int main()
{
    map< string, string > trans_map;
    typedef map< string, string >::value_type valType;

    // 第一个权宜之计: 将转换对固定写在代码中
    trans_map.insert( valType( "gratz", "grateful" ));
    trans_map.insert( valType( "'em", "them" ));
    trans_map.insert( valType( "cuz", "because" ));
    trans_map.insert( valType( "nah", "no" ));
    trans_map.insert( valType( "sez", "says" ));
    trans_map.insert( valType( "tanx", "thanks" ));
    trans_map.insert( valType( "wuz", "was" ));
    trans_map.insert( valType( "pos", "suppose" ));
```

单词转换map

- 程序执行时产生如下输出

```
Here is our transformation map:

key: 'em value: them
key: cuz value: because
key: gratz value: grateful
key: nah value: no
key: pos value: suppose
key: sez value: says
key: tanx value: thanks
key: wuz value: was

Here is our original string vector:

nah I sez tanx cuz I wuz pos
to not cuz I wuz gratz

Here is our transformed string vector:
no I says thanks because I was suppose
to not because I was grateful
```

从map 中删除元素

- erase()函数
  - 可以传递一个键值、一个iterator或一对iterator

```
map<string,loc*>::iterator where;
where = text_map.find( removal_word );

if ( where == text_map->end() )
    cout « "oops: " « removal_word « " not found!\n";
else {
    text_map->erase( where );
    cout « "ok: " « removal_word « " removed!\n";
}
```

本节内容

- 生成文本位置map
- 创建单词排除集
- 完整的程序
- multimap 和multiset
- 栈
- 队列和优先级队列

定义set 并放入元素

- set 只是键的集合

```
#include <set>
set<string> exclusion_set;
```

- 用insert 操作将单个元素插入到set 中

```
exclusion_set.insert( "the" );
exclusion_set.insert( "and" );
```

  - 可以向insert()提供一对iterator 以便插入一个元素序列

- copy()：对元素赋值

搜索元素

- find()
  - 如果元素存在则返回指向这个元素的iterator
  - 否则返回一个等于end()的iterator，表示该元素不存在
- count()
  - 如果找到元素返回1
  - 如果元素不存在则返回0

```
if ( exclusion_set.count( textword ))
continue;
// ok: 把单词加入到map 中
```

迭代set中的对象

- set 只能含有每个键值的惟一实例

```
// 获得指向位置向量的指针
loc *ploc = (*text_map)[ query_text ];
// 对 "位置项对" 进行迭代
// 把每行插入到set 中
set< short > occurrence_lines;
loc::iterator liter = ploc->begin(),
                liter_end = ploc->end();
while ( liter != liter_end ) {
    occurrence_lines.insert( occurrence_lines.end(), (*liter).first );
    ++liter;
}
```

迭代set中的对象

- occurrence_line 保证只包含单词出现行的单实例，为了显示这些文本行只需迭代set：

```
register int size = occurrence_lines.size();
cout << “\n” << query_text << " occurs " << size
        << (size == 1 << " time:" : " times:“)<< "\n\n";
set< short >::iterator it=occurrence_lines.begin();
for ( ; it != occurrence_lines.end(); ++it ) {
        int line = *it;
        cout << "\t( line “<< line + 1 << " ) “<< (*text_file)[line] << endl;
}
```

- set 支持操作size()、empty()和erase()
  - 同map 类型相同

本节内容

- 生成文本位置map
- 创建单词排除集
- 完整的程序
- multimap 和multiset
- 栈
- 队列和优先级队列

完整的程序

- C Primer 3th Edition 中文版
  Stanley B. Lippman, Josee Lajoie 著
  潘爱民 张丽译

本节内容

- 生成文本位置map
- 创建单词排除集
- 完整的程序
- multimap 和multiset
- 栈
- 队列和优先级队列

multimap 和multiset

- multiset 和multimap 允许要被存储的键出现多次

```
#include <map>
multimap< key_type, value_type > multimapName;
// 按string 索引, 存有list<string>
multimap< string, list< string > > synonyms;

#include <set>
multiset< type > multisetName;
```

multimap 和multiset

- 迭代策略
  - 联合使用由find()返回的iterator 和由count()返回的值

```
#include <map>
#include <string>
void code_fragment()
{
    multimap< string, string > authors;
    string search_item( "Alain de Botton" );
    // ...
    int number = authors.count( search_item );
    multimap< string,string >::iterator iter;
    iter = authors.find( search_item );
    for ( int cnt = 0; cnt < number; ++cnt, ++iter )
      do_something( *iter );
    // ...
}
```

multimap 和multiset

- 迭代策略
- 使用equal_range()，返回一对iterator，如果这个值存在，则
  - 第一个iterator 指向该值的第一个实例
  - 第二个iterator 指向这个值的最后实例的下一位置

```
#include <map>
#include <string>
#include <utility>
void code_fragment()
{
    multimap< string, string > authors;
    // ...
    string search_item( "Haruki Murakami" );
    while ( cin && cin >> search_item )
      switch ( authors.count( search_item ))
      {
        // 不存在, 继续往下走
        case 0:
          break;
```

multimap 和multiset

- 插入和删除操作与关联容器set 和map 相同

- equal_range()可以提供iterator 对，因此可以标记出要被删除的多个元素的范围

```
#include <multimap>
#include <string>
typedef multimap< string, string >::iterator iterator;
pair< iterator, iterator > pos;
string search_item( "Kazuo Ishiguro" );
// authors 是一个 multimap<string,string>
// 这等价于 authors.erase( search_item );
pos = authors.equal_range( search_item );
authors.erase( pos.first, pos.second );
```

multimap 和multiset

- 每次插入增加一个元素例如

```
typedef multimap<string,string>::value_type valType;
multimap<string,string> authors;
// 引入Barth 下的第一个键
authors.insert( valType(string( "Barth, John" ),string( "Sot-Weed Factor" )));
// 引入Barth 下的第二个键
authors.insert( valType(
string( "Barth, John" ),
string( "Lost in the Funhouse" )));
```

- 不支持下标操作是访问multimap 元素的一个限制例如

```
authors[ "Barth, John" ]; // 错误: multimap
```

- 将导致编译错误

本节内容

- 生成文本位置map
- 创建单词排除集
- 完整的程序
- multimap 和multiset
- 栈
- 队列和优先级队列

栈

- 栈容器(stack container)：先进后出

操作	功能
empty()	如果栈为空，则返回true，否则返回false
size()	返回栈中元素的个数
pop()	删除，但不返回栈顶元素
top()	返回，但不删除栈顶元素
push(item)	放入新的栈顶元素

栈


```
#include <stack>
#include <iostream>
int main()
{
        const int ia_size = 10;
        int ia[ia_size ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        // 填充 stack
        int ix = 0;
        stack< int > intStack;
        for ( ; ix < ia_size; ++ix )
                intStack.push( ia[ ix ] );
        int error_cnt = 0;
        if ( intStack.size() != ia_size ) {
                cerr << "oops! invalid intStack size: “<< intStack.size()<< "\t expected: " << ia_size << endl;
                ++error_cnt;
        }
        int value;
        while ( intStack.empty() == false )
        {
                // 读取栈顶元素
                value = intStack.top();
                if ( value != --ix ) {
                        cerr << "oops! expected " << ix<< " received " << value<< endl;
                        ++error_cnt;
                }
                // 弹出栈顶元素, 并重复
                intStack.pop();
        }
        cout << "Our program ran with “<< error_cnt << " errors!" << endl;
}
```

栈

- 栈类型被称为容器适配器container adapter
  - 它把栈抽象施加在底层容器集上
  - 缺省情况下栈用容器类型deque 实现
  - 因为deque为容器前端的插入和删除提供了有效支持

- 要改写这种缺省的实现，可以提供显式的顺序容器类型作为第二个参数

```
stack< int, list<int> > intStack;
```

- 栈的元素被按值输入每个对象被拷贝到底层的容器中
  - 一种取代的存储策略是定义一个指针栈

```
#include <stack>
class NurbSurface { /* mumble */ };
stack< NurbSurface* > surf_Stack;
```

本节内容

- 生成文本位置map
- 创建单词排除集
- 完整的程序
- multimap 和multiset
- 栈
- 队列和优先级队列

队列和优先级队列

- 队列queue
  - 先进先出FIFO 即first in first out 的存储和检索策略

- 标准库提供了两种风格的队列
  - FIFO 队列就称作queue
  - priority_queue 优先级队列
    - 允许用户为队列中包含的元素项建立优先级
    - 没有把新元素放在队列尾部而是放在比它优先级低的元素前面

队列和优先级队列

操作	功能
empty()	如果队列为空，则返回true，否则返回false
size()	返回队列中元素的个数
pop()	删除，但不返回队首元素，在priority_queue中，队首元素代表优先级最高的元素
front()	返回，但不删除队首元素，只能应用在一般队列上
back()	返回，但不删除队尾元素，只能应用在一般队列上
top()	返回，但不删除priority_queue的优先级最高的元素，只能应用在priority_queue上
push(item)	在队尾放入一个新元素，对于priority_queue，将根据优先级排序

Reference

- C Primer 3th Edition 中文版
  Stanley B. Lippman, Josee Lajoie 著
  潘爱民 张丽译

Thank You!