



第六章

string 类 与 STL 简介

内容安排



- 6.1 标准库string类型
- 6.2 标准模板库STL
- 6.3 综合实例





6.1 标准库string类型

C语言：

- 存放字符串利用数组，如char a[20], char *s= “abcd” ;
- 字符串的处理通过调用系统提供的库函数
- 把数据和对数据的处理分离开，不符合面向对象程序设计的要求

- C++标准类库中预定义了一个字符串类，即string类。
- string类中封装字符串的概念，并提供了许多关于字符串操作的成员函数。
- 使用string类需要包含头文件string（不是string.h）
- 注意：string类型与char *、char 数组[]是不同的数据类型，不能从string类型到char *或数组的类型自动转换或赋值。





6.1 标准库string类型

- 6.1.1 string类的构造函数
- 6.1.2 string类的部分成员函数
- 6.1.3 string类的运算符





6.1.1 string类的常用构造函数

常用构造函数的原型	功 能
<code>string()</code>	默认构造函数，创建一个长度为0的字符串
<code>string(const string & rhs)</code>	拷贝构造函数，利用已经存在的字符串来创建新的字符串
<code>string(const string & rhs, unsigned pos,unsigned n)</code>	将已经存在的字符串从第pos的位置开始，取n个字符来创建新的字符串
<code>string(const char*s)</code>	利用字符数组s来创建新的字符串
<code>string(const char*s,unsigned n)</code>	利用字符数组s的前n个字符来创建新的字符串
<code>string(unsigned n,char c)</code>	将字符c重复n次，来创建新的字符串

● 【例6-1】利用string类提供的各种构造函数来创建字符串对象。



```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    char * s = "Hello world.";
    string s1;    //默认构造函数，空串
    cout<<"s1 = "<<s1<<endl;
    string s2(s);    //构造函数string(const char * s)
    cout<<"s2 = "<<s2<<endl;
    string s3(s2);    //拷贝构造函数
    cout<<"s3 = "<<s3<<endl;
    string s4(s2,1,3); //将s2从第1的位置开始，取3个字符创建s4
    cout<<"s4 = "<<s4<<endl;
    string s5(s,3); //利用s的前3个字符来创建s5
    cout<<"s5 = "<<s5<<endl;
    string s6(6,'H'); //将字符H重复6次，创建S6
    cout<<"s6 = "<<s6<<endl;
    return 0;
}
```

6.1.2 string类的部分成员函数

成员函数的原型	功 能
<code>string& append(const char*s)</code> <code>string& append(const char*s ,unsigned n)</code> <code>string& append(const string &str,unsigned pos, unsigned n)</code>	将字符串s添加到本字符串的尾部 将字符串s的前n个字符添加到本字符串的尾部 将字符串str从第pos的位置开始的n个字符添加到本字符串的尾部
<code>string& assign(const string &str, unsigned pos, unsigned n)</code>	将字符串str从第pos的位置开始的n个字符赋值给本字符串
<code>int compare (const string & str) const</code>	将本字符串与str比较，若本字符串小于str，则返回负数；若本字符串大于str，则返回正数；若本字符串等于str，则返回0
<code>unsigned find(const string &str, unsigned pos=0) const</code>	在本字符串中查找str，若找到了，则返回str第一次出现的位置；若没有找到，则返回 string::npos
<code>string& insert(unsigned p0,const string &str, unsigned pos, unsigned n)</code>	将字符str从pos位置开始的n个字符插入到本字符串p0的位置处
<code>unsigned length() const</code>	返回本字符串的长度
<code>string& replace(unsigned p0,unsigned n0, const string &str)</code>	用字符串str替换本字符串从p0开始的n0个字符
<code>unsigned size() const</code>	返回本字符串的大小
<code>string substr (unsigned pos=0,unsigned n=np0) const</code>	返回从pos开始的n个字符构成的字符串
<code>void swap(string &str)</code>	将本字符串与字符串str进行交换



串查找与string::npos值

- string 类提供了 6 种形式的查找函数,每种函数以不同find函数名称命名, 用于查找一个字符串是否包含另外一个子字符串。

find: 从前往后查找子串或字符出现的位置。

rfind: 从后往前查找子串或字符出现的位置。

find_first_of: 从前往后查找何处出现另一个字符串中包含的字符。

find_last_of: 从后往前查找何处出现另一个字符串中包含的字符。

find_first_not_of: 从前往后查找何处出现另一个字符串中没有包含的字符。

find_last_not_of: 从后往前查找何处出现另一个字符串中没有包含的字符





- 对字符串的操作，如：插入、查找、求子串等，常基于字符的位置进行，**注：串中字符位置从0计数，即串的第1个字符的序号为0、第2个字符的位置为1、依此类推。**
- 查找函数都返回 `string::size_type` 类型的结果值（**实际是unsigned类型**）。
- 如果找到了（一个字符串中包含另外一个字符串），则返回找到的起始位置。
- 如果没有找到，则返回一个名为 **`string::npos`** 的特殊值，写程序时，可利用该值判断是否找到。比如：


```
string str;  
  
string::size_type pos=str.find_first_of("h");  
  
if(pos!=string::npos)  
{..  
  
....  
  
}
```






- 【例6-2】利用string类提供的成员函数来完成字符串的插入操作。





```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string str("He world.");
    string ins;
    cout<<"原字符串是: "<<str<<endl;
    cout<<"输入要插入的字符串: ";
    cin>>ins;
    str.insert(2,ins,0,ins.size());
    cout<<"新字符串是: "<<str<<endl;
    return 0;
}
```



6.1.3 string类的运算符



运算符	示例	功 能
+	A+B	合并A和B（有一个操作数是string类型即可，另一个可以是” ”）
=	A=B	将B赋值给A
+=	A+=B	A=A+B，合并A和B，然后赋给A
==,!=,<,<=,>,>=	A==B,A!=B, A<B, A<=B, A>B, A>=B	比较A和B
[]	A[i]	A中的第i个元素
<<	cout<<A	将A插入到输出流对象，即输出A
>>	cin>>A	从输入流对象提取字符串赋给A，输入A

● 【例6-3】利用string类提供的运算符来完成字符串的排序操作。



```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string str[5];
    cout<<"输入5个字符串: \n";
    for(int i=0;i<5;i++)
        cin>>str[i];
    cout<<"排序前, 这5个字符串为: \n";
    for(int i=0;i<5;i++)
        cout<<str[i]<<"\n";
    for(int i=0;i<4;i++)
        for(int j=i+1;j<5;j++)
            if(str[i]>str[j])
            {
                string temp;
                temp = str[i];
                str[i] = str[j];
                str[j] = temp;
            }
}
```

```
    cout<<"排序后, 这5个字符串为: \n";
    for(int i=0;i<5;i++)
        cout<<str[i]<<"\n";
    return 0;
}
```

● 练习: 从键盘上输入2个字符串,
将他们拼接在一起输出。





```
#include <iostream> //练习
#include <string>
using namespace std;
int main()
{
    string s1,s2,s3;
    cout<<“请输入2个字符串: ”<<endl;
    cin>>s1;
    cin>>s2;
    s3=s1+s2;
    cout<<“拼接后的字符串: ”<<s<<endl;
    return 0;
}
```





(一) string字符串的输入输出

- 1、string字符串的常规输入、输出

- (1) cin (不能读取带空格的字符串)

- (2) getline(cin, string变量);用来读取包含空格的字符串;

- (3) cout输出



(一) string字符串的输入输出

● 2、string 字符串的流输入、输出：

(1) 使用流对象stringstream、istringstream 和 ostringstream类，可以将 string 对象当作一个流进行输入输出。标准库定义了三种类型的字符串流：

- **istringstream**，由 **istream** 派生而来，提供**读 string** 的功能。
- **ostringstream**，由 **ostream** 派生而来，提供**写 string** 的功能。
- **stringstream**，由 **iostream** 派生而来，提供**读写 string** 的功能。

(2) 使用这两个类需要包含头文件 sstream，即#include<sstream>

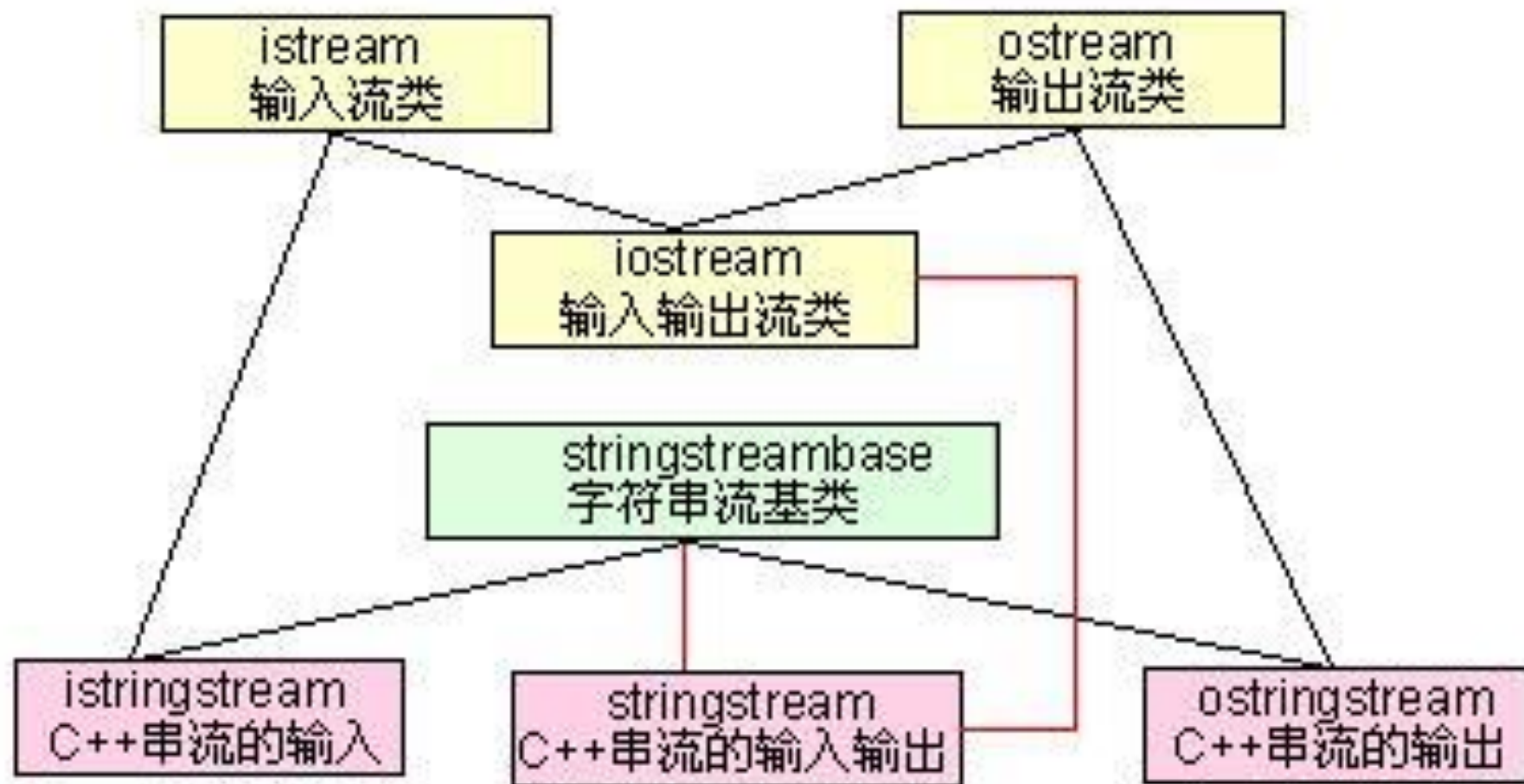
(3) 从istringstream输入流对象进行读取，过程和从 cin 读取一样，只不过输入的来源由键盘变成了输入流对象。

将输出结果输出到流ostringstream对象，而不是屏幕，用 ostringstream 类的 str 成员函数能将输出到 ostringstream 对象中的内容提取出来。

stringstream类同时具有读写功能，功能分别与上述类似。



(一) string字符串的输入输出





```
#include<iostream>
#include<sstream>
using namespace std;
int main()
{
    istringstream iss("12 34");
    int a, b;
    iss >> a >> b; //从字符串输入流iss中读取两个数
    cout << a << " " << b << endl;
    ostringstream oss; //输出到string str中
    oss << a << " " << b;
    cout << oss.str() << endl; //取出输出缓冲区的值
    return 0;
}
```





(二) 针对string 对象中单个字符处理函数

(1) 要对string对象中的单个字符进行处理，如某个特殊字符是否为空格字符、字母、数字等等，可以使用以下各种字符操作函数，适用于string对象的字符（或其他任何char值），这些函数定义在cctype头文件中。

(2) 取出string对象中的单个字符，可以利用下标完成，下标序号从0开始。





- `isalnum(c)` //判断字符变量c是否为字母或数字
- `isalpha(c)` //判断字符c是否为英文字母
- `iscntrl(c)` //判断字符c是否为控制字符
- `isdigit(c)` //检查c是否为阿拉伯数字0到9
- `islower(c)` //检查c是否为小写英文字母
- `isupper(c)` //检查c是否为大写英文字母
- `ispunct(c)` //检查c是否为标点符号或特殊符号
- `isspace(c)` //检查c是否为空格字符
- `tolower(c)` //如果c为大写字母，则返回小写字母；否则返回原来的值
- `toupper(c)` //如果c为小写字母，则返回大写字母；否则返回原来的值

课堂练习：把一个字符串对象转换为全大写





```
#include <string>  
#include<iostream>  
using namespace std;  
int main()  
{  
    string str("hello");  
    for(string::size_type index = 0; index != str.size(); ++index)  
        if(islower(str[index]))  
            str[index] = toupper(str[index]);  
    cout<<str<<endl;  
    return 0;  
}
```





6.2 标准模板库STL

- 6.2.1 标准模板库
- 6.2.2 容器
- 6.2.3 迭代器
- 6.2.4 泛型算法





6.2.1 标准模板库

- STL (Standard Template Library, 标准模板库) 是一套功能强大的 C++ 类模板, 提供了通用的类模板和函数模板, 它们可以实现多种常用的算法和数据结构, 如向量、链表、队列、栈等。
- STL采用泛型编程, 为C++的程序员提供了一个可扩展的应用框架, 高度体现了软件的可复用性。
- STL 是 C++ 标准库的一部分, 不用单独安装。



6.2.1 标准模板库



- (一) STL的3个主要组件及其关系：

- **container**（容器）：容器是用来管理某一类对象的集合，通过其提供的方法，保存各种类型的节点元素，比如 **deque**、**list**、**vector**、**map** 等。
- **algorithm**（算法）：算法作用于容器。它们提供了执行各种操作的方式，包括对容器内容执行初始化、排序、搜索和转换等操作，不需要关注容器的内部细节。
- **iterator**（迭代器）：迭代器是一种行为类似指针的对象，它指向的对象为容器中元素（结构体）的节点，用于遍历访问容器或其子集中的对象元素。每个容器都有专属的迭代器，而算法通过迭代器对容器中的元素进行操作。

- 容器和算法通过迭代器可以进行无缝连接。





- **(二) STL发展简介:**
- STL是C++标准库的一个重要组成部分，它由Alexander Stepanov、Meng Lee和David R Musser等人最先开发，它是与C++几乎同时开始开发的；
- 最初STL是以Ada为实现语言，后以C++作为其实现语言，后STL又被添加到C++库。
- 1996年，惠普公司免费公开了STL，为STL的推广做了很大的贡献。
- STL体现了范型化程序设计的思想，使用STL编写的程序具有高度的可重用性和可移植性，而且程序的效率也很高。
- 程序员不用思考具体实现过程，只要能够熟练的应用就可以了，这样可以把更多的精力放在程序开发的别的方面。





- (三) STL有关的头文件：
- STL中的全部成员都定义在名字空间std中。
- STL被组织成头文件的形式来提供给程序员，例如：
 - <vector>是“向量”容器的头文件（不同的容器用不同的头文件）
 - <iterator>是迭代器的头文件
 - <algorithm>是算法的头文件等



6.2.2 容器



- **容器**是一种存储了有限集合数据元素的数据结构。
- STL 容器可用来存放、容纳各种不同类型的数据，它们都是**类模板**。
- 实例化为类型T 的容器类能够存放T 类型的对象。
- 基本容器有7个：
 - 向量 (**vector**)
 - 双端队列 (**deque**, 用得不多)
 - 列表 (**list**)
 - 集合 (**set**)
 - 多重集合 (**multiset**)
 - 映射 (**map**)
 - 多重映射 (**multimap**)

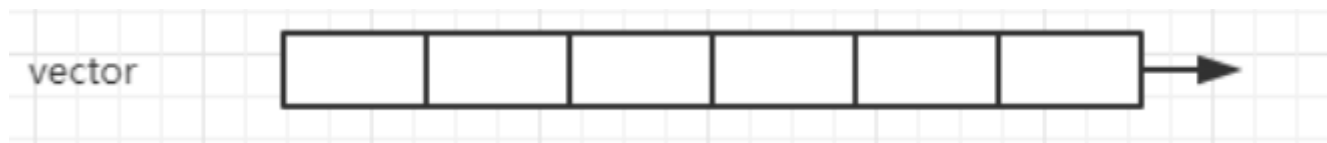


6.2.2 容器



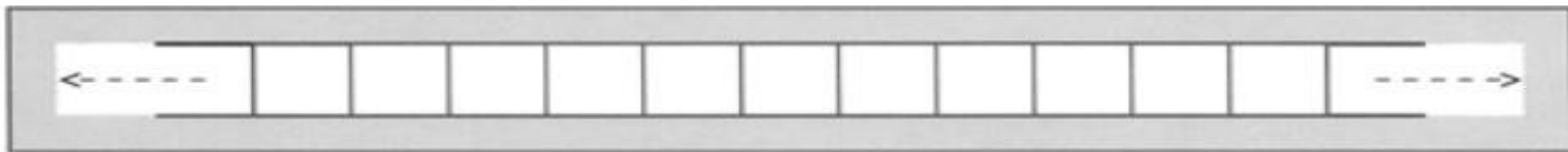
1、向量 (vector)

用一块连续的内存空间来保存数据，是可变大小的序列容器（大小可变的数组），能够在插入或删除元素时自动调整其大小。初始内存空间大小可以预先指定，当存储的数据超过分配的空间时，vector会重新分配一块内存。



2、双端队列 (deque, 用得不多)

deque是一种优化了的对序列两端元素进行添加和删除操作的基本序列容器。结构上是一个动态数组。



6.2.2 容器

3、列表 (list)

list是一种序列式容器，其结构是一个双向链表，其中的数据元素是通过链表指针串连成逻辑意义上的线性表。

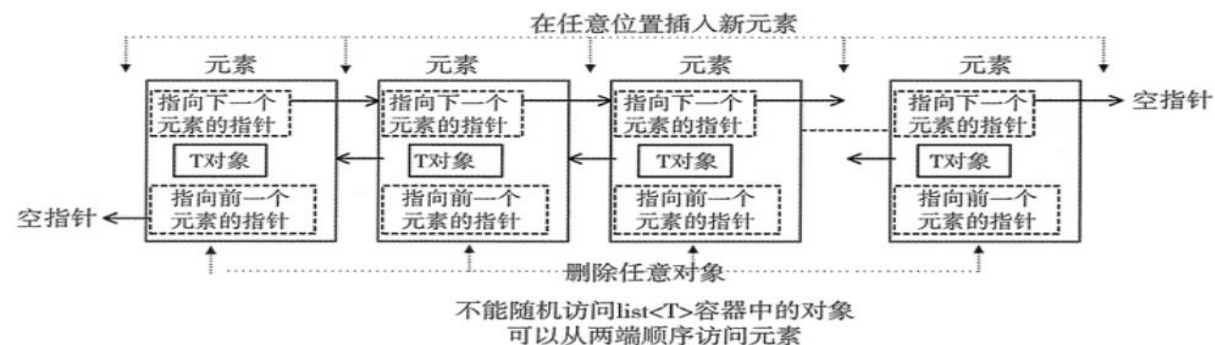
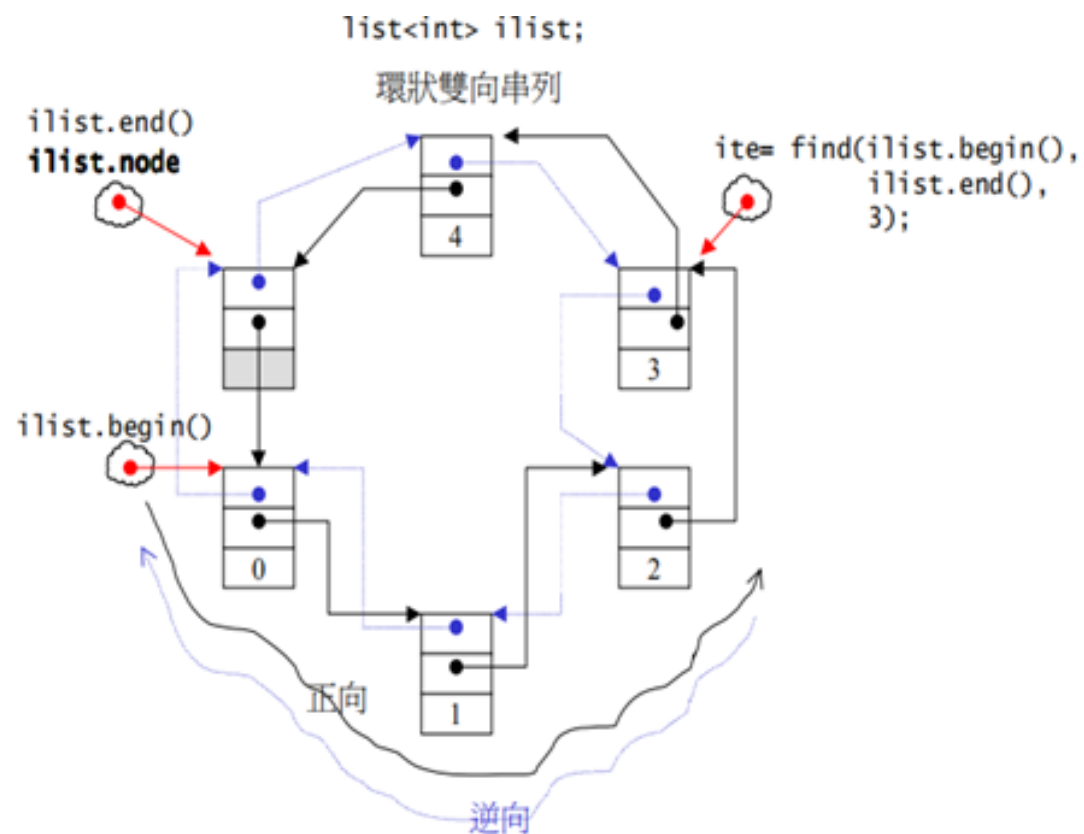


图 1 list 双向链表容器的存储结构示意图



6.2.2 容器



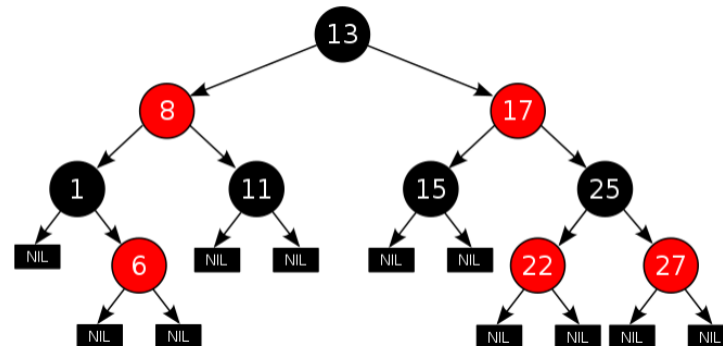
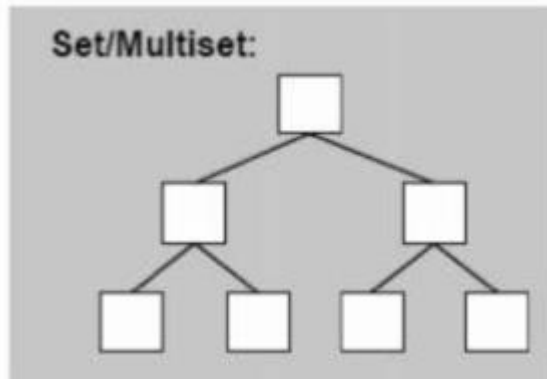
4、集合 (set)

属于关联式容器，使用一颗红黑树的非线性数据结构，每一次插入数据都会自动进行排序，因此，set中的元素总是有序的，这些数据的值(value)必须是唯一的。

即：数据自动进行排序且数据唯一，是一种集合元素，允许进行数学上的集合相关的操作。

5、多重集合 (multiset)

multiset是一种元素值 可复的、有序的关联容器。底层结构为二叉搜索树(红黑树)。multiset元素的值不能在容器中进行修改(因为元素 总是const的)，但可以从容器中插入或删除。存储的是<key, value>的键值对。



6.2.2 容器



6、映射 (map)

Map是一个关联容器，它通过键值对<key,value>的方式管理数据，map内部自建一颗红黑树(一种非严格意义上的平衡二叉树)，这颗树具有对数据自动排序的功能，集合中的元素按一定的顺序排列。map 中 key 值是唯一的，每个键只能出现一次，元素插入过程是按排序规则插入，所以不能指定插入位置。即**一对一映射**。

7、多重映射 (multimap)

multimap 是关联容器的一种，multimap 也是键值对<key,value>的方式管理数据，容器中的元素是按关键字排序的，并且允许有多个元素的关键字相同，即multimap 中相同键可以出现多次。即**一对多映射**。





容器	描述	所在头文件
vector 顺序容器	连续存储的元素（推荐使用）	<vector>
deque	连续存储的指向不同元素的指针所组成的数组	<deque>
list	由节点组成的双向链表，每个结点包含着一个元素	<list>
set 关联容器	由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序	<set>
multiset	和set基本相同，但允许存在两个次序相等的元素的集合	<set>
map	由{键，值}对组成的集合，以某种作用于键对上的谓词排列	<map>
multimap	允许键对有相等的次序的映射，即一个键可以对应多个值	<map>



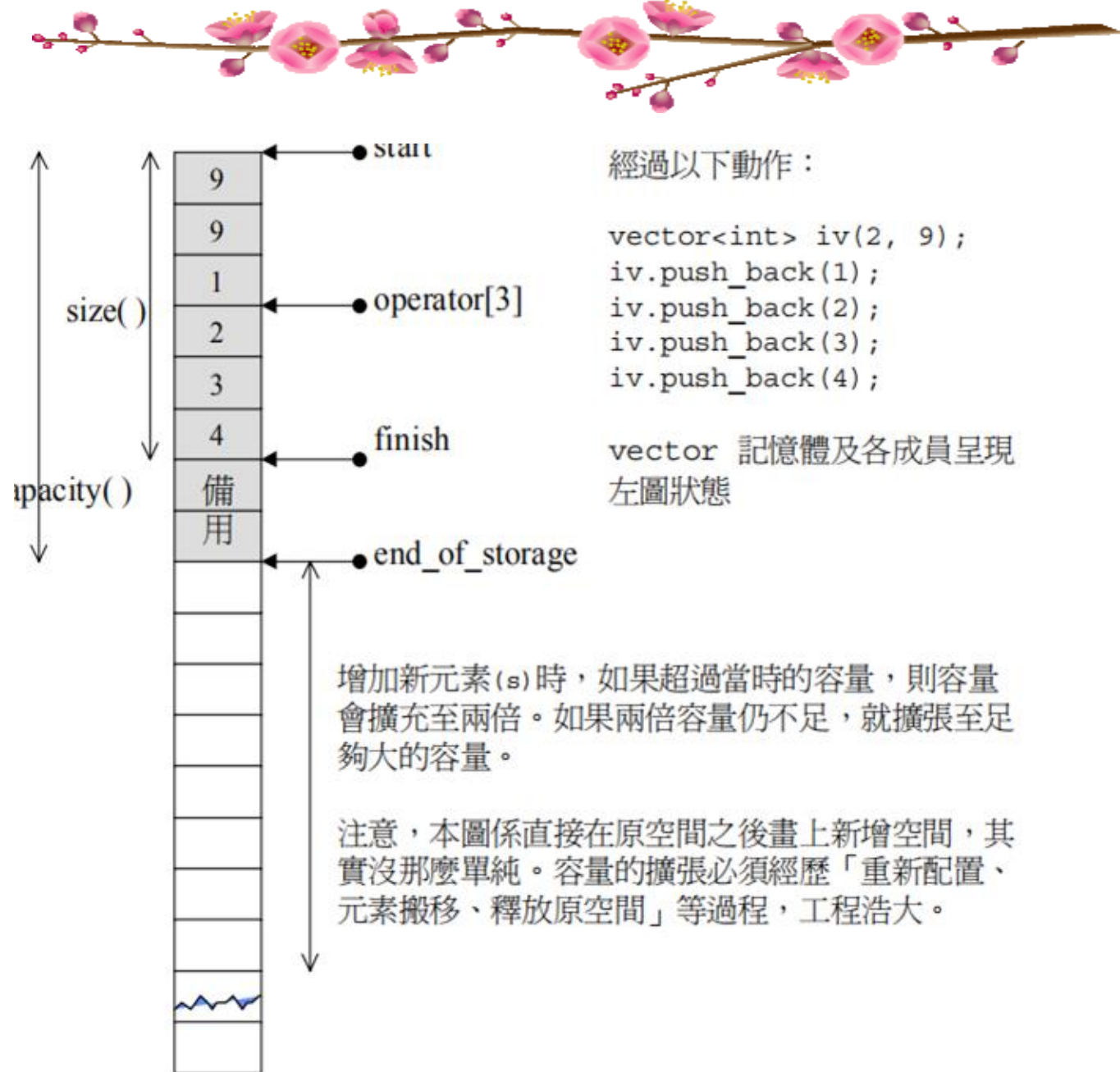
- 在数据元素的组织方式上有所不同：
 - 顺序容器是将装入其中的对象按照严格的线性形式组织起来，对其的访问可以是顺序的，也可以是随机的；
 - 关联容器是将装入其中的对象按照非线性的形式组织起来，可以根据一组索引来快速的访问对象。
- vector和deque的逻辑结构相类同；
- set和multiset的逻辑结构相类同；
- map和multimap的逻辑结构相类同。
- vector和list的主要差别是：
 - **vector**是用动态数组实现的；
 - **list**使用链表实现的。



典型容器使用举例：

(1) vector容器及使用

- vector容器与C++中的数组相类同，它是一个模板类，因此能够包含任何类型的数据，vector的内存存储空间是连续的，开始时先申请一特定大小的空间，当空间用完之后会进行扩容，扩容之后，再将原内存空间中的内容拷贝到新的内存空间中，最后把原来的空间释放。



vector容器中常用的成员函数如表所示。



成员函数	功 能
vector()	默认构造函数，创建一个大小为0的空向量
vector (size_type n,const T &val=T())	创建一个大小为n的向量，每个元素的初始值为val，默认为T构造的元素
size_type size () const	返回当前容器中存放的元素的个数
size_type capacity () const	返回给当前分配给向量容器中最多可以容纳的元素的个数
bool empty () const	如果当前容器为空则返回true，否则返回false
void push_back (const T&x)	在容器的尾部添加元素x
void pop_back ()	删除容器中的最后一个元素
iterator insert (iterator it, const T&x)	在迭代器it所指的容器位置前插入元素x
iterator erase (iterator it)	删除容器中迭代器it所指向的元素
void clear ()	删除容器中的所有元素
iterator begin ()	返回指向第一个元素的迭代器
iterator end ()	返回指向最后一个元素的后一个位置的迭代器
T front ()	返回容器中的第一个元素
T back ()	返回容器中的最后一个元素
void reserve (size_type n)	重新分配更多的内存，使其可以存储n个元素





说明:

(1) **vector**容器是使用类型参数化方式(泛型编程)设计的,其成员函数中的元素类型为T, T的具体数据类型由定义对象时进行实例化。如:

```
vector<int> vec1;           //定义了一个整型的向量容器的对象vec1
```

(2) **size_type**是无符号整型。

● 【例6-4】vector容器的使用方法。

● 参 考：vector & iterator



● `#include <iostream>`

● `#include <vector>`

● `using namespace std;`

● `int main()`

● `{`

● `vector<int> vec1(3, 6);` //定义了一个整型的向量容器的对象vec1，有3个元素，每个元素初值为6

● `cout << "vec1中的元素个数: " << vec1.size() << "\n";`//size函数返回当前容器中的元素个数3

● `cout << "vec1中的元素值分别为: ";`

● `for (vector<int>::size_type i = 0; i<vec1.size(); i++)`

● `cout << vec1[i] << ' ';` //通过下标方式访问容器中的元素

● `cout << "\n\n";`

● `for (vector<int>::iterator it1 = vec1.begin(); it1 != vec1.end(); it1++)`

● `cout << *it1 << ' ';` //通过迭代器访问容器中的元素

● `cout << "\n\n";`

```
vector<char> vec2;           //定义一个char的向量容器的对象vec2，大小为0
vec2.push_back('C');        //在容器尾部添加元素 “C”
vec2.push_back('h');        //在容器尾部添加元素 “h”
vec2.push_back('n');        //在容器尾部添加元素 “n”
vec2.push_back('a');        //在容器尾部添加元素 “a”
vector<char>::iterator it;   //定义字符型向量容器的迭代器对象it
it = vec2.begin() + 2;      //指向第3个元素
vec2.insert(it, 'i');       //在it指向的位置前插入'i'
cout << "vec2中的元素个数: " << vec2.size() << "\n";
cout << "vec2中的第一个元素值为: " << vec2.front() << "\n";
cout << "vec2中的元素值分别为: ";
for (vector<char>::size_type i = 0; i<vec2.size(); i++)
    cout << vec2[i] << " ";
cout << endl;
return 0;
}
```

典型容器使用举例： (2) list容器及使用

list容器的常用成员函数：

- push_front(x): 把元素x推入（插入）到头部
- push_back(x): 把元素x推入（插入）到尾部
- pop_front(): 弹出（删除）第一个元素
- pop_back(): 弹出（删除）最后一个元素
- begin(): 返回向量中第一个元素的迭代器
- rbegin()返回指向第一个元素的逆向迭代器
- end()返回末尾的迭代器
- rend() 指向list末尾的逆向迭代器
- front(): 获得list容器中的头部元素
- back(): 获得list容器的最后一个元素。
- erase()删除迭代器指向的一个元素
- remove()删除匹配值的元素(匹配元素全部删除)
- remove_if()删除条件满足的元素(遍历一遍列表)
- unique()删除list中重复的元素

- clear(): 清空list中的所有元素。
- assign()给list赋值
- insert()插入一个元素到list中
- max_size()返回list能容纳的最大元素数量
- size()返回list中的元素个数
- empty(): 利用empty() 判断list是否为空。
- resize()改变list的大小
- reverse()把list的元素倒转
- merge() 合并两个list
- splice()合并两个list
- swap()交换两个list
- sort()给list排序



典型容器使用举例：

(2) list容器及使用



list容器的构造函数

list的构造函数与vect的构造函数相类似：

- list() 构造空的列表
- list (size_type n, const value_type& val = value_type()) 构造并初始化n个val的列表
- list (const list& x) 拷贝构造，利用一个列表创建另外一个列表
- template <class InputIterator> list (InputIterator first, InputIterator last)

使用已有列表的迭代器所指定的元素进行初始化构造另一个列表

比如：

```
list<int> c0; //空链表
```

```
list<int> c1(3); //建一个含三个默认值是0的元素的链表
```

```
list<int> c2(5, 2); //建一个含五个元素的链表，值都是2
```

```
list<int> c4(c2); //建一个c2的copy链表
```

```
list<int> c5(c1.begin(), c1.end()); c5含c1一个区域的元素[_First, _Last)。
```



典型容器使用举例：

(2) list容器及使用



list容器的sort成员函数（排序），共有4种情况：

- list中的元素为基本数据类型（比如，整数等）时：

- (1) 可用不带参数的sort()函数，默认按升序排列；

- (2) 如需要按降序排列，则需要自定义比较函数，并将该函数传递给sort函数或将标准库函数对象greater<Type>()传递给sort函数（functional头文件中已对库函数greater进行了定义）。



典型容器使用举例：

(2) list容器及使用



- list中的元素类型为自定义的结构体类型或者类类型时：

(1) 必须在类里友元函数形式重载>或<操作符（分别表示升序或降序），并将标准库函数对象greater<Type>()传递给sort函数（functional头文件中已对库函数greater进行了定义）。

(2) 自定义一个类，并在该类中重载()运算符函数，最后将该类对象传递给sort函数。



典型容器使用举例：

(2) list容器及使用



list容器的元素访问注意事项：

- $p1[i]$ ：不能通过下标访问 list 容器中指定位置处的元素。
- $p1-=i$ 、 $p1+=i$ 、 $p1+i$ 、 $p1-i$ ：双向迭代器 $p1$ 不支持 $-=$ 、 $+=$ 、 $+$ 、 $-$ 运算符。
- $p1 < p2$ 、 $p1 > p2$ 、 $p1 \leq p2$ 、 $p1 \geq p2$ ：双向迭代器 $p1$ 、 $p2$ 不支持使用 $<$ 、 $>$ 、 \leq 、 \geq 比较运算符。



● list示例1



```
● #include<iostream>//list1
● #include<list>
● using namespace std;
● template <typename T>
● struct greate{
●     bool operator()(const T & t1, const T & t2) //自定义排序函数
●     {
●         if (t1 > t2)          //'>'降序
●             return true;
●         return false;
●     }
● };
● int main()
● {
●     //定义链表
●     list<int> list1;
●     //从链表尾部加入元素
●     for (int i = 6; i <= 10; i++)
●         list1.push_back(i);
```



- `//从链表头部加入元素`
- `for (int i = 5; i >= 0; i--)`
- `list1.push_front(i);`
- `//获取链表元素个数`
- `cout<<list1.size()<<endl;`
- `//顺序输出链表各个元素`
- `for (it1 = list1.begin(); it1 != list1.end(); it1++)`
- `cout<<*it1<<" ";`
- `cout<<"\n";`
- `//分别从头部和尾部各删除一个元素`
- `list1.pop_front();`
- `list1.pop_back();`





- //删除第1个元素

- **list1.erase(list1.begin());**

- //删除链表中匹配值的元素

- **list1.remove(8);**

- **for (it1 = list1.begin(); it1 != list1.end(); it1++)**

- **cout<<*it1<<" ";**

- **cout<<"\n"; // 2 3 4 5 6 7 9**

- //往链表里插入元素的三种方法

- **list1.insert(++list1.begin(), 20);** //在第1个数后插入元素20

- **list1.insert(list1.begin(), 5, 30);**//在最前面插入5个元素，值均为30

- **for (it1 = list1.begin(); it1 != list1.end(); it1++)**

- **cout<<*it1<<" ";**

- **cout<<"\n"; // 30 30 30 30 30 2 20 3 4 5 6 7 9**



//list2是list1的copy

list<int> list2(list1);

list1.insert(++list1.begin(), list2.begin(), list2.end()); **//把list2从头到尾插入到list1第1个数后面**

for (it1 = list1.begin(); it1 != list1.end(); it1++)

cout<<*it1<<" ";

cout<<"\n";**// 30 30 30 30 30 30 2 20 3 4 5 6 7 9 30 30 30 30 2 20 3 4 5 6 7 9**

//逆序输出链表

list<int>::reverse_iterator it2;

for (it2 = list1.rbegin(); it2 != list1.rend(); it2++)

cout<<*it2<<" ";

cout<<"\n";

//链表反转

list1.reverse();

for (it1 = list1.begin(); it1 != list1.end(); it1++)

cout<<*it1<<" ";

cout<<"\n";





- //链表排序，默认升序
- **list1.sort();**
- **for (it1 = list1.begin(); it1 != list1.end(); it1++)**
- **cout<<*it1<<" ";**
- **cout<<"\n";**
- **list1.sort(greate<int>());** //降序
- **for (it1 = list1.begin(); it1 != list1.end(); it1++)**
- **cout<<*it1<<" ";**
- **cout<<"\n";**
- //删除相邻重复的元素
- **list1.unique();**
- **for (it1 = list1.begin(); it1 != list1.end(); it1++)**
- **cout<<*it1<<" ";**
- **cout<<"\n";**
- **list<int> list3;**
- **for (int i = 1; i <= 4; i++)**
- **list3.push_back(i * 100); // 100 200 300 400**
- **list1.sort(); // merge需要排序**





```
//把list3合并到list1，并使之有序，默认升序
list1.merge(list3);    //合并后list3为空
for (it1 = list1.begin(); it1 != list1.end(); it1++)
    cout<<*it1<<" ";
cout<<"\n";
for (int i = 1; i <= 4; i++)
    list3.push_back(i * 100);
list1.sort(greate<int>()); //降序
list3.sort(greate<int>()); //降序
list1.merge(list3, greate<int>());    //降序，合并后list3为空
for (it1 = list1.begin(); it1 != list1.end(); it1++)
    cout<<*it1<<" ";
cout<<"\n";
//清除list中的所有元素
list1.clear();
list2.clear();
return 0;
}
```



list示例2



- `#include <iostream> //第2个示例`
- `#include <list>`
- `#include <functional>`
- `using namespace std;`
- `#define N 1010`
- `// 调用操作符和函数对象`
- `class Node{`
- `public:`
- `int num;`
- `friend bool operator>(const Node & t1, const Node & t2){ return t1.num > t2.num; }`
- `} node[N];`
- `class S{`
- `public: bool operator()(const Node& t1, const Node& t2) //自定义排序函数，重载函数调用操作符()`
- `{ if (t1.num > t2.num) //‘>’降序`
- `return true;`
- `return false;`
- `};`

- `int main()`
- `{`
- `int f[6] = { 2, 3, 4, 5, 1, 0 };`
- `list<Node> list1;`
- `for (int i = 0; i<6; i++){`
- `node[i].num = f[i];`
- `list1.push_back(node[i]); }`
- `// S s1; //函数对象`
- `// list1.sort(s1); // 降序`
- `// list1.sort(S()); //构造一个临时函数对象`
- `//使用标准库函数对象greater<Type>`
- `list1.sort(greater<Node>());`
- `list<Node>::iterator ite;`
- `for (ite = list1.begin(); ite != list1.end(); ite++)`
- `cout<<ite->num<<" ";`
- `cout<<"\n";`
- `return 0;`
- `}`
- `//标准库函数对象的调用 sort(svec.begin(),svec.end(),greater<string>());`
- `//greater<string>()是传递比较元素的谓词函数，是一个greater<string>类型的临时函数对象`

6.2.4 迭代器



1、迭代器的概念：

- 迭代器是一种能对容器内元素遍历访问的**指针数据类型**。
- 迭代器既能对容器中的对象进行访问，又能将容器的结构隐藏起来，达到封装的效果。
- 迭代器类型提供了比下标更通用的方法（只有少数容器支持下标）。
- 它提供了灵活的访问形式，支持很多种运算方式（不同类型的迭代器支持的运算不完全一致），如：

①可以使用运算符“++”或“--”，让迭代器前后移动

②可以使用运算符<、<=、>、>=、==、!=等比较两个迭代器相对位置

③可以使用+、+=、-、-=使迭代器在序列中前进（或后退）给定整数个元素后的位置

④可以用“*迭代器”的形式来访问迭代器所指向的元素。



6.2.4 迭代器

2、迭代器的种类：

迭代器种类 (Category)	能力	提供者
Output 迭代器	向前写入	Ostream, inserter
Input 迭代器	向前读取一次	Istream
Forward 迭代器	向前读取	Forward list, unordered containers
Bidirectional 迭代器	向前和向后读取	List, set, multiset, map, multimap
Random-access 迭代器	以随机访问方式读取	Array, vector, deque, string, C-style array

迭代器是连接容器和算法的一种重要桥梁。



3、容器的iterator类型（迭代器）

- 每种容器都定义了各自的迭代器类型，如vector:

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

表示定义了一个名为iter的变量，它的类型是由vector<int>定义的iterator类型。

- 所有的容器都定义有自己的iterator类型，因此如果单单使用容器，只需要包含对应容器的头文件即可。
- 有些特殊的iterator，被定义在头文件<iterator>中。

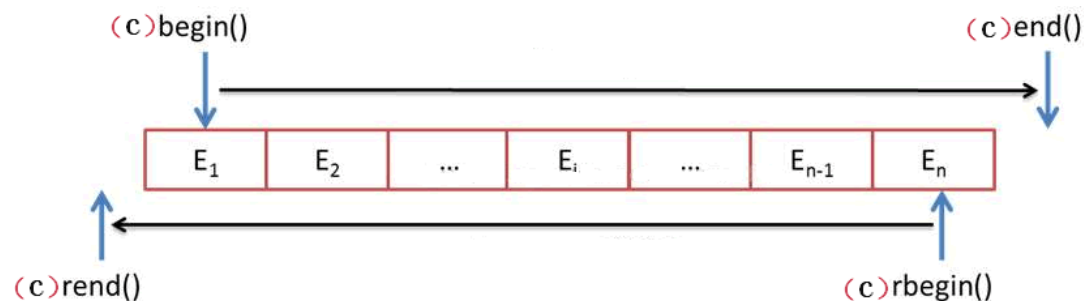
表1：不同容器的迭代器的功能

容器	迭代器功能
vector	随机访问
deque	随机访问
list	双向
set / multiset	双向
map / multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器



4、容器的begin()、end()成员函数及使用

- 每种容器都定义了一对名为begin和end的成员函数，用于返回迭代器
- begin()返回的迭代器指向第一个元素
- end()返回的迭代器指向末端元素的下一个
- 如果容器为空，begin和end返回相同
- 迭代器可使用*,++,--, 比如 *iter, iter++, iter-- （设iter为迭代器变量）
- 也可使用 == , !=
- 也可以 iter + n , iter - n
- 循环条件里面使用end()时要注意循环体是否有添加或插入操作；



● 【例6-5】用迭代器操作vector容器的元素。

- `#include <iostream>`
- `#include <vector>`
- `using namespace std;`
- `int main()`
- `{`
- `vector<int> vec1(5); //定义一个整型向量容器对象vec1, 有5个元素`
- `vector<int>::iterator it1; //定义整型向量容器的迭代器对象it1`
- `cout<<"输入5个整数: ";`
- `for(it1 = vec1.begin();it1!=vec1.end();it1++)`
- `cin>>*it1;`
- `cout<<"vec1中的元素个数: "<<vec1.size()<<endl;`
- `cout<<"vec1中的元素值分别为: ";`
- `for(it1 = vec1.begin();it1!=vec1.end();it1++)`
- `cout<<*it1<<" ";`
- `cout<<"\n\n";`
- `vector<char> vec2(5); //定义一个char向量容器对象vec2, 有5个元素`
- `vector<char>::iterator it2; //定义char向量容器的迭代器对象it2`
- `cout<<"输入5个字符: ";`



- `for(it2 = vec2.begin();it2!=vec2.end();it2++)`
- `cin>>*it2;`
- `cout<<"vec2中的元素个数: "<<vec2.size()<<endl;`
- `cout<<"vec2中的元素值分别为: ";`
- `for(it2 = vec2.begin();it2!=vec2.end();it2++)`
- `cout<<*it2<<" ";`
- `cout<<endl;`
- `return 0;`
- `}`





5、输入、输出迭代器

- STL中定义了输入迭代器`istream_iterator`、输出迭代器`ostream_iterator`，这类迭代器可与输入或输出流绑定在一起，用于迭代遍历所关联的I/O流。
- 使用这些迭代器，不仅可以方便地完成键盘的输入和屏幕的输出，而且可以方便地进行文件的操作。
- 使用输入、输出迭代器，需要嵌入 `iterator` 头文件：`#include <iterator>`



6、输入、输出迭代器的构造函数



输入迭代器istream_iterator:

(1) istream_iterator是iterator的一个公有派生类模板，用于从一个输入流中提取元素。

(2) 输入迭代器对象的定义格式:

① `istream_iterator<T> in(strm);`

表示创建从输入流strm读取T类型对象的istream_iterator对象。

如: `istream_iterator<double> iit(cin);` 创建一个从标准输入流 cin 读取数据的输入流迭代器

② `istream_iterator<T> in;`

表示创建出一个具有结束标志的输入流迭代器istream_iterator对象，即输入流中遇到非T类型的数据时即终止。



输出迭代器ostream_iterator:



(1) `ostream_iterator`也是`iterator`的一个公有派生类模板，用于从一个把元素插入到输出流中。

(2) 输出迭代器对象的定义方式:

① `ostream_iterator<T> out(strm);` 创建将T类型的对象写到输出流`strm`的`ostream_iterator`对象，如: `ostream_iterator<int> out_it(cout);`创建了一个可将 `int` 类型元素写入到输出流（屏幕）中的迭代器。

② `ostream_iterator<T> out(strm, delim);` 创建将T类型的对象写到输出流`strm`的`ostream_iterator`对象，在写入过程中使用`delim`作为元素的分隔符，如: `ostream_iterator<int> out_it(cout, ",");`创建输出流迭代器，写入 `int` 类型元素的同时，还会附带写入一个逗号（,）。

● 【例6-6】用输入、输出迭代器操作vector容器的元素。



- **#include <iostream>**
- **#include <vector>**
- **#include <iterator>**
- **using namespace std;**
- **//copy函数的参数都是迭代器**
- **// #include <algorithm>**
- **//output_iterator copy(input_iterator start, input_iterator end, output_iterator dest);**
- **//The copy function copies the elements between start and end to dest.**
- **//In other words, after copy has run,**
- **//*dest = *start *(dest+1) = *(start+1) *(dest+2) = *(start+2) ... *(dest+N) = *(start+N)**
- **int main()**
- **{**
- **istream_iterator<int> is(cin);**
- **istream_iterator<int> eof;**
- **ostream_iterator<int> out(cout, " ");**
- **vector<int> text;**



- // 将标准输入的内容复制至text中。 由于使用的是vector，故使用back_inserter()
- `copy(is, eof, back_inserter(text));` //输入若干个整数，以非数字结束
- `cout<<"输入的"<<text.size()<<"个整数分别为: ";`
- `copy(text.begin(),text.end(),out);`
- `cout<<endl;`
- `return 0;`
- `}`

注:

- (1) back_inserter: 创建一个使用push_back的迭代器
- (2) copy函数: `std::copy(start, end, container);`是用来复制任何具有迭代器的对象的元素，start，end是需要复制的源文件的头地址和尾地址，container是接收器的起始地址。



- **#include <iostream> //另一种方法**
- **#include <vector>**
- **#include <iterator>**
- **using namespace std;**

- **//copy函数的参数都是迭代器**
- **// #include <algorithm>**
- **//output_iterator copy(input_iterator start, input_iterator end, output_iterator dest);**
- **//The copy function copies the elements between start and end to dest.**
- **//In other words, after copy has run,**
- **//*dest = *start *(dest+1) = *(start+1) *(dest+2) = *(start+2) ... *(dest+N) = *(start+N)**

- **int main()**
- **{**
- **typedef vector<int> int_vector;**
- **typedef istream_iterator<int> input;**
- **typedef ostream_iterator<int> output;**

```
//插入适配器，它的作用是引导copy算法每次在容器末端插入一个数据
typedef back_insert_iterator<int_vector> ins;
int_vector vec;
cout<<"输入若干个整数，以非数字结束：";
//三个参数分别为起始位置、结束位置和目的地
//前两个参数是两个迭代器的临时对象，前一个指向整型输入数据流的开始，后一个则指向"pass-the-end
value"。
//第一个迭代器从开始位置每次累进，最后到达第二个迭代器所指向的位置
copy(input(cin),input(),ins(vec));
cout<<"输入的"<<vec.size()<<"个整数分别为：";
//output(cout, " ")展开后的形式是： ostream_iterator(cout, " "), 其效果是产生一个处理输出数据流的迭代器//对象，
其位置指向数据流的起始处，并且以" "作为分割符。copy函数将会从头至尾将vector中的内容
//“拷贝”到输出设备，第一个参数所代表的迭代器将会从开始位置每次累进，最后到达第二个参数所代表//的迭代
器所指向的位置。
copy(vec.begin(),vec.end(),output(cout, " "));
cout<<endl;
return 0;
}
```




● 【例6-7】利用输入、输出迭代器将一个文件中的内容复制到另一个文件中。



- **#include <vector>**
- **#include <iostream>**
- **#include <iterator>**
- **#include <fstream>**
- **#include <string>**
- **using namespace std;**

- **int main()**
- **{**
- **ifstream file1("file1.txt");**
- **ofstream file2("file2.txt");**
- **if(!file1||!file2)**
- **{**
- **cout<<"cannot open file\n";**
- **return 0;**
- **}**
- **}**

- **istream_iterator<string> in(file1);**
- **istream_iterator<string> eof;**
- **ostream_iterator<string> out(file2," ");**
- **ostream_iterator<string> out1(cout," ");**
- **vector<string> text;**
- **// 将标准输入的内容复制至text中。 由于使用的是**
//vector, 故使用back_inserter()
- **copy(in, eof, back_inserter(text));**
- **copy(text.begin(),text.end(),out);**
- **copy(text.begin(),text.end(),out1);**
- **return 0;**
- **}**

```
● #include <iostream> //另一种方法
● #include <fstream>
● #include <string>
● #include <vector>
● #include <iterator>
● using namespace std;
● int main()
● {
●     typedef vector<string> string_vector;
●     typedef istream_iterator<string> input;
●     typedef ostream_iterator<string> output;
●     typedef back_insert_iterator<string_vector>
ins;
●     string_vector vec;
●     ifstream from_file("file1.txt");
●     if(from_file.fail()) {
●         cout<<"file1.txt打开失败"<<"\n";
●         exit(0);
●     }
●     copy(input(from_file),input(),ins(vec));
●     copy(vec.begin(),vec.end(),output(cout,"\n"));
●     cout<<endl;
●     ofstream to_file("file2.txt");
●     copy(vec.begin(),vec.end(),output(to_file,"*"));
●     return 0;
● }
```

6.2.5 泛型算法



- STL的算法库中，包含了超出100种通用算法，这些算法均被设计成函数模板，因此具有通用性。
比如：
 - `find`用于查找在容器中是否存在等于某个特定值的元素；
 - `sort`用于对容器中的元素进行排序等。`sort`函数是快速排序算法的实现函数。语句 `sort(vec.begin(), vec.end());` 的功能是将vector容器对象vec中从开始位置到结束位置的所有数据元素进行排序(升序)。
- 大多数算法通过遍历由两个迭代器标记的一段元素来实现其功能。这些算法除了用在容器上之外，也可以用于内置数组类型等。
- 使用算法库中的算法（函数模板）应包含头文件<algorithm>。



6.2.5 泛型算法



- `sort`函数用于C++中，对给定区间所有元素进行排序，默认为升序，也可进行降序排序。`sort`函数包含在头文件为`#include<algorithm>`的c++标准库中。
- 语法: `sort(start, end, cmp)`
- (1) `start`表示要排序数组的起始地址;
- (2) `end`表示数组结束地址的下一位;
- (3) `cmp`用于规定排序的方法，可不填，默认升序。

注意:

- `vector`的迭代器是随机访问迭代器，支持泛型算法的`sort`及其算法。可以使用STL中的`sort`。
- `list`容器上的迭代器是双向的，不支持随机访问，因此不能使用需要随机访问迭代器的`sort`算法。`list`容器使用自己的成员函数`sort`排序。





- sort函数有三个参数：
- （1）第一个是要排序的数组的起始地址。
- （2）第二个是结束的地址（最后一位要排序的地址）
- （3）第三个参数是排序的方法，可以是从小到大也可能是从大到小，还可以不写第三个参数，此时**默认的排序方法是从小到大排序**





sort用于基本数据类型数组示例

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

int main()
{
    int a[] = { 5, 4, 3, 2, 1 };

    sort(a, a + 5);

    for (int i = 0; i < 5; i++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

```
#include <iostream>
#include <algorithm>
using namespace std;
bool cmp(int x,int y)
{
    return x>y;
}
int main()
{
    int a[] = { 5, 4, 3, 2, 1 };

    sort(a, a + 5,cmp);

    for (int i = 0; i < 5; i++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```



- 【例6-8】设计一个程序，从键盘上输入若干个整型数据，利用算法库中的排序函数进行排序，并将排序后的结果输出。





- `#include <vector>`
- `#include <iostream>`
- `#include <iterator>`
- `#include <algorithm>`
- `using namespace std;`
- `bool cmp(int x, int y)`
- `{`
- `return x>y;`
- `}`
- `int main()`
- `{`
- `cout<<"输入一组整数，以非数字结束:\n";`
- `istream_iterator<int> in(cin);`
- `istream_iterator<int> eof;`
- `ostream_iterator<int> out(cout," ");`
- `vector<int> text;`
-





- `copy(in, eof, back_inserter(text));`
- `cout << "排序前的" << text.size() << "个整数分别为: ";`
- `copy(text.begin(),text.end(),out);`
- `cout<<endl;`
- `sort(text.begin(),text.end());`
- `cout << "从小到大排序后的" << text.size() << "个整数分别为: ";`
- `copy(text.begin(),text.end(),out);`
- `cout<<endl;`
- `sort(text.begin(),text.end(),cmp);`
- `cout << "从大到小排序后的" << text.size() << "个整数分别为: ";`
- `copy(text.begin(),text.end(),out);`
- `cout<<endl;`
- `return 0;`
- `}`



迭代器、算法和容器的关系



1、三者之间联系：

每个容器都有专属的迭代器，而算法通过迭代器对容器中的元素进行操作。

2、容器

容器能够通过模板的方法，装下各种类型的节点元素。

3、迭代器是一种smart pointer

4、算法：

算法通过操作容器对应的迭代器，就可以间接地操作容器中的元素。而不需要关注容器的内部细节





6.3 综合实例

- 【例6-9】设计一个程序，从键盘上输入若干个字符串，利用算法库中的查找函数对给定的字符串进行查找，并将查找后的结果输出。
- 分析：根据本题的题意：
 - 首先定义1个string类型的vector容器，并创建相应的对象；
 - 然后从键盘输入流中提取若干个字符串放到vec容器中；调用算法库中的查找函数对给定的字符串进行查找。
 - 最终最后将查找的结果输出。





- **#include <iostream>**
- **#include <string>**
- **#include <vector>**
- **#include <iterator>**
- **#include <algorithm>**
- **using namespace std;**
- **int main()**
- **{**
- **vector<string> vec(10);**
- **typedef ostream_iterator<string> output;**
- **cout << "输入若干个字符串，以-1结束:";**
- **int i;**
- **for (i = 0; i < (int)vec.size(); i++)**
- **{**
- **cin >> vec[i];**
- **if (vec[i] == "-1")**
- **break;**
- **}**





- `cout << "输入的" << i << "个字符串分别为: ";`
- `copy(vec.begin(), vec.begin() + i, output(cout, " "));`
- `cout << "\\n";`
- `string str;`
- `cout << "输入要查找的字符串";`
- `cin >> str;`
- `vector<string>::iterator it;`
- `it = find(vec.begin(), vec.begin() + i, str);`
- `if (it != vec.begin()+i)`
- `cout << "该字符串存在" << *it<<endl;`
- `else`
- `cout << "该字符串不存在" << endl;`
- `return 0;`
- `}`





6.4 小结

- (1) 对于字符串，可以使用string类的各种构造函数来完成其初始化的工作；可以使用string类提供的丰富的成员函数和运算符进行其相关的操作。
- (2) 标准模板库STL是最新的C++标准库中的一个子集，这个庞大的子集占据了整个库的大约80%的分量。而作为在实现STL过程中扮演关键角色的模板则充斥了几乎整个C++标准库。
- (3) STL主要包含了容器、迭代器和算法3个部分。
 - 容器是一组元素的集合；
 - 算法库中的**100**多种算法覆盖了相当大的应用领域；
 - 迭代器的作用是能够让容器与算法不干扰的相互发展，最后又能无间隙的粘合起来。

